A Semantics-based Approach to Querying Process Model Repositories

Liang Song[1,2,3,4], Jianmin Wang[1,3,4], Arthur H.M. ter Hofstede[5,6,7], Marcello La Rosa[5,7], Chun Ouyang[5,7], and Lijie Wen[1,3,4]

[1] School of Software, Tsinghua University, Beijing, China
{songliang08}@mails.thu.edu.cn, jianmin@mail.thu.edu.cn
[2] Department of Computer Science and Technology, Tsinghua University, Beijing, China
[3] Key Lab for Information System Security, Ministry of Education, Beijing, China
[4] National Laboratory for Information Science and Technology, Beijing, China
[5] Queensland University of Technology, Brisbane, Australia
{a.terhofstede,m.larosa,c.ouyang}@qut.edu.au
[6] Eindhoven University of Technology, Eindhoven, The Netherlands
[7] NICTA Queensland Lab, Australia

Abstract. As business process management technology matures, organisations acquire more and more business process models. The resulting collections can consist of hundreds, even thousands of models and their management poses real challenges. One of these challenges concerns model retrieval where support should be provided for the formulation and efficient execution of business process model queries. As queries based on only structural information cannot deal with all querying requirements in practice, there should be support for queries that require knowledge of process model semantics. In this paper we formally define a process model query language that is based on semantic relationships between tasks. This query language is independent of the particular process modelling notation used, but we will demonstrate how it can be used in the context of Petri nets by showing how the semantic relationships can be determined for these nets in such a way that state space explosion is avoided as much as possible. An experiment with three large process model repositories shows that queries expressed in our language can be evaluated efficiently.

1 Introduction

With the increasing maturity of business process management, more and more organisations need to manage large numbers of business process models, and among these may be models of high complexity. Business process models constitute valuable intellectual property capturing the way an organisation conducts its business. Processes may be defined along the entire value chain and over time a business may gather hundreds and even thousands of business process models. As an example consider Suncorp, one of the largest Australian insurers. Over the years, Suncorp have gone through a number of organizational mergers and acquisitions, as a result of which the company has accumulated over 3,000 process models for the various lines of insurance. In this context, support for business process retrieval, e.g. for the purposes of process reuse or process standardization, is a challenging proposition.

While several query languages exist that can be used to retrieve process models from a repository, e.g. BPMN-Q [1] or BP-QL [2, 3], these languages are based on syntactic relationships between tasks and not on semantic relationships between them.

While in a process graph, a task A may follow a task B this does not mean that during execution task B will always follow sometime after task A. Let us consider for example the two process models in Fig. 1. These models represent two variants of a business process for opening bank accounts in the BPMN notation [4]. Each variant consists of a number of tasks (represented by rectangles) and dependencies between these tasks. Arcs represent sequential dependencies, while diamonds represent decisions (if there is one incoming arc and multiple outgoing arcs) and simple merges (if there are multiple incoming arcs and one outgoing arc). These two variants could capture the way an account is opened in two different states in which the company operates, and could be part of a collection of various process models for all states in which the bank operates. Now let us assume that an analyst needs to find all states which require an assessment of the customer's credit history when opening an account. In this case, by only using the structural relationships between tasks, we cannot discern between the two variants, i.e. we would retrieve them both, since in both models there is at

least a path from task “Receive customer request” to task “Analyse customer credit history”. However, if we consider semantic relationships, we can see that task “Analyse customer credit history” always follows task “Receive customer request” in all instances of the first process variant, but this is not the case for one instance of the second variant (the one where task “Open VIP account” is run). Thus we can correctly exclude the second variant from the results of our query, and return the first variant only.

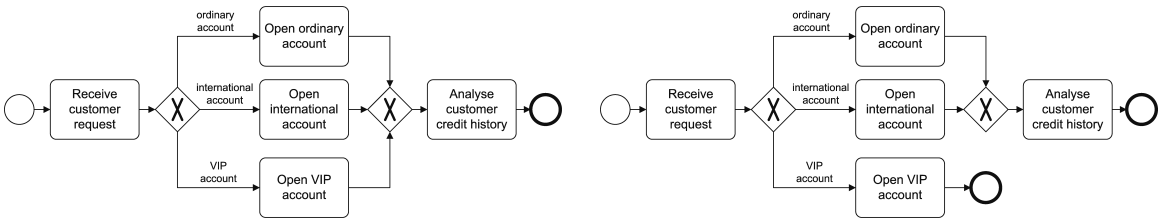

Fig. 1: Two variants of a business process for opening bank accounts.

A process model retrieval language based on semantic relationships is indeed in line with process execution, e.g. through a workflow management system, and such a language may therefore be more intuitive to use by stakeholders who are not necessarily modelling experts. One challenge though, when it comes to determining semantic relationships between tasks, is how to determine these relationships in a feasible manner, i.e. without suffering from the well-known state space explosion problem.

In light of the above, in this paper we aim to address the development of an *expressive* business process model query language. We do so by proposing a new query language for process model repositories, namely “A Process-model Query Language” (APQL). This language relies on a number of basic temporal relationships between tasks which can be composed to obtain complex relationships between them. These predicates allow us to express queries that can discriminate over single process instances or task instances.

Since the language relies on temporal relationships between tasks, it is independent of a specific business process modelling notation. However, in order to demonstrate its feasibility, we provide a concrete realization of this language in the context of Petri nets. To this end, we adopt the theory of *complete finite prefixes* [5],and its improvements [6],to compute temporal relationships between Petri net transitions.

Moreover, we implemented a tool for evaluating APQL queries over repositories of Petri net models, and used this tool to evaluate the performance of the approach over three process model collections (two from practice, the third, a much larger one, artificially generated). The performance measurements show that indeed this technique can efficiently cope with complex queries based on process semantics, issued over very large repositories.

The remainder of this paper is organized as follows. In Section 2 we formally define both syntax and semantics of APQL while in Section 3 we show how this language can be operationalized for Petri nets. Next, in Section 4 we present a tool implementation based on this realisation in Petri nets and report on the performance evaluation of this tool. Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

2 The Business Process Model Query Language APQL

In this section we introduce *APQL* (A Process-model Query Language). First we provide an informal introduction to the language together with its abstract syntax (Section 2.1), then we provide a formal semantics (Section 2.3).

2.1 Syntax

In this section we look at the design rationale for APQL and provide an informal introduction to this language. The syntax of APQL is presented in the form of an abstract syntax, the advantages of which over a concrete syntax have been espoused by Meyer [7]. In essence, in an abstract syntax we can avoid committing ourselves prematurely to specific choices for keywords or to the order of various statements.

APQL is designed as a process model retrieval language that is independent of the actual process modelling language used. This is important as in practice a variety of modelling languages is used (e.g. BPMN, EPCs) and the language should be generally applicable. Another important starting point is the fact that process models have a semantics and it should be possible to exploit this semantics when querying. For example, let t_i (where $i = 1, 2, ..., n$) be a task identifier, while task t_1 may follow task t_2 in process model r (i.e. there is a directed path from task t_1 to task t_2), it may be the case that due to the presence of certain splits and joins, task t_2 cannot actually be executed after task t_1. Hence, a language based on the syntax (i.e. structure) of a process model is not always powerful enough.

In order to achieve language independence we define a set of 20 basic predicates to capture, in business process models, the occurrences of tasks as well as the semantic relationships between tasks. Below, the first two predicates capture the occurrence of a task in some or every execution of a given process model.

1. $posoccur(t_1, r)$: there exists some execution of process model r where at least one instance of t_1 occurs.
2. $alwoccur(t_1, r)$: in every execution of process model r, at least one instance of t_1 occurs.

The next two predicates capture the exclusive and concurrent relationships between task occurrences. Note that these two predicates do not assume that an instance of t_1 or t_2 should eventually occur.

3. $exclusive(t_1, t_2, r)$: in every execution of process model r, it is never possible that an instance of t_1 and an instance of t_2 both occur.
4. $concur(t_1, t_2, r)$: t_1 and t_2 are not causally related, and in every execution of process model r, if an instance of t_1 occurs then an instance of t_2 occurs and vice versa.

Then we consider various forms of causal relationship between task occurrences. The relationship can be *precedence* or *succession*, where one task may occur *immediately* or *eventually* preceding or succeeding another task. It may hold for *any* or *every* occurrence of the tasks in *some* or *every* process execution. Combining all these considerations results in 16 forms of causal relationships which are captured by the remaining 16 basic predicates as follows.

Let Φ be one of the following intermediate predicates,

1. $succ_{any}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and is eventually succeeded by an instance of t_2 (e.g. $...t_1...t_2...$).
2. $succ_{every}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and every instance of t_1 is eventually succeeded by an instance of t_2 (e.g. $t_1...t_1...t_2$).
3. $pred_{any}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and is eventually preceded by an instance of t_2 (e.g. $...t_2...t_1...$).
4. $pred_{every}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and every instance of t_1 is eventually preceded by an instance of t_2 (e.g. $t_2...t_1...t_1$).
5. $isucc_{any}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and is immediately succeeded by an instance of t_2 (e.g. $...t_1t_2...$).
6. $isucc_{every}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and every instance of t_1 is immediately succeeded by an instance of t_2 (e.g. $t_1t_2...t_1t_2$).
7. $ipred_{any}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and is immediately preceded by an instance of t_2 (e.g. $...t_2t_1...$).
8. $ipred_{every}(t_1, t_2, i)$: in process execution i, at least one instance of t_1 occurs and every instance of t_1 is immediately preceded by an instance of t_2 (e.g. $t_2t_1...t_2t_1$).

then

- $\Phi^{\forall}(t_1, t_2, r)$: $\Phi(t_1, t_2, i)$ holds *for every* process execution i of process model r, and
- $\Phi^{\exists}(t_1, t_2, r)$: there *exists* some process execution i of process model r where $\Phi(t_1, t_2, i)$ holds.

In this paper we will demonstrate how these relations can be computed for Petri nets, but when a realisation is provided for another language then APQL can also be used for querying process model collections where the process models are specified according to that language. Note that these predicates are all defined in terms of the execution of a process. Below, we define the abstract syntax of APQL using the notation introduced in [7].

In APQL a query is a set of *Assignments* combined with a *Predicate*.

$$\begin{aligned} Query &\triangleq s : Assignments; p : Predicate \\ Assignments &\triangleq Assignment^* \end{aligned}$$

The result is those process models that satisfy the *Predicate*. An *Assignment* assigns a *TaskSet* to a variable and when evaluating the *Predicate* every variable is replaced by the corresponding *TaskSets*.

$$Assignment \triangleq v : Varname; ts : TaskSet$$

*TaskSet*s can be enumerations of tasks or they can be defined in terms of other *TaskSet*s either by *Construction* or by *Application*. A *TaskSet* can also be defined through a variable, a *TaskSetVar*.

$$TaskSet \triangleq SetofTasks \mid Construction \mid Application \mid TaskSetVar$$

A *Task* can be defined as a combination of a *TaskLabel* and a *SimDegree*. The idea is that one may be interested in *Tasks* of which the task label bears (at least) a certain degree of similarity to a given activity name. There are a number of definitions in the literature concerning label similarity and for a concrete implementation of the language one has to commit to one of these.

$$\begin{aligned} SetofTasks &\triangleq Task^+ \\ Task &\triangleq TaskLabelExpr \\ TaskLabelExpr &\triangleq l : TaskLabel; d : SimDegree \end{aligned}$$

A *TaskSetVar* is simply a variable that may be used in assignments.

$$TaskSetVar \triangleq v : Varname$$

A *TaskSet* can be composed from other *TaskSet*s through the application of the well-known set operators such as *union*, *difference*, and *intersection*. Another way to construct a *TaskSet* is by the application of a *TaskCompOp* (i.e. one of the basic predicates introduced earlier, but now interpreted as a function) on another *TaskSet*. In that case we have to specify whether we are interested in the tasks that have that particular relation with *all* or with *any* of the tasks in the *TaskSet*. For example, an expression with *TaskSet* S, *TaskCompOp* *PosSuccAny* and with *AnyAll* indicator *all*, should yield those tasks that any instance of such a task succeeds an instance of each individual task in S in some process execution.

$$\begin{aligned} Construction &\triangleq ts_1, ts_2 : TaskSet; o : Set\_Op \\ Set\_Op &\triangleq Union \mid Difference \mid Intersection \\ Application &\triangleq ts : TaskSet; o : TaskCompOp; a : AnyAll \\ TaskCompOp &\triangleq Exclusive \mid Concur \mid \\ &\quad AlwSuccAny \mid AlwSuccEvery \mid AlwPredAny \mid AlwPredEvery \mid \\ &\quad PosSuccAny \mid PosSuccEvery \mid PosPredAny \mid PosPredEvery \mid \\ &\quad AlwISuccAny \mid AlwISuccEvery \mid PosISuccAny \mid PosISuccEvery \mid \\ &\quad AlwIPredAny \mid AlwIPredEvery \mid PosIPredAny \mid PosIPredEvery \\ AnyAll &\triangleq Any \mid All \end{aligned}$$

A *Predicate* can consist of a simple *Task*, with the intended semantics that all process models containing that *Task* should be retrieved, a *TaskAlw*, with the intended semantics what the basic predicate *exists* specifies, a *TaskRel*, with the intended semantics that all process models satisfying that particular relation should be retreived, or it can be recursively defined as a binary or unary *Predicate* through the application of logical

operators.

$$
\begin{array}{rcl}
Predicate & \triangleq & Task \mid TaskAlw \mid TaskRel \mid Bin\_Predicate \mid Un\_Predicate \\
Bin\_Predicate & \triangleq & o : BinLogOp; p_1, p_2 : Predicate \\
Un\_Predicate & \triangleq & o : UnLogOp; p : Predicate \\
BinLogOp & \triangleq & And \mid Or \\
UnLogOp & \triangleq & Not \\
TaskAlw & \triangleq & t : Task
\end{array}
$$

A *TaskRel* can be 1) a relationship between a *Task* and a *TaskSet* checking whether that *Task* occurs in that *TaskSet* (*TaskInTaskSet*), 2) a relationship between a *Task* and a *TaskSet* and involving a *TaskCompOp* and an *AnyAll* indicator determining whether the *Task* has the *TaskCompOp* relationship with any/all *Tasks* in the *TaskSet* (*Task_TaskSet*), 3) a relationship between two *Tasks* involving a *TaskCompOp* predicate determining whether for the two *Tasks* that predicate holds (*Task_Task*), 4) a relationship between two *TaskSets* involving a *TaskCompOp* and an *AnyAll* indicator determining whether the *Tasks* in those *TaskSets* all have that *TaskCompOp* relationship to each other or whether for each *Task* in the first *TaskSet* there is a corresponding *Task* in the second *TaskSet* for which the relationship holds (*Elt_TaskSet_TaskSet*), or 5) a relationship between two *TaskSets* determined by a set comparison operator (*Set_TaskSet_TaskSet*).

$$
\begin{array}{rcl}
TaskRel & \triangleq & w : TaskRelExpr \\
TaskRelExpr & \triangleq & TaskInTaskSet \mid Task\_TaskSet \mid \\
 & & Task\_Task \mid Elt\_TaskSet\_TaskSet \mid \\
 & & Set\_TaskSet\_TaskSet \\
TaskInTaskSet & \triangleq & t : Task; ts : TaskSet \\
Task\_TaskSet & \triangleq & t : Task; ts : TaskSet; \\
 & & o : TaskCompOp; a : AnyAll \\
Task\_Task & \triangleq & t_1, t_2 : Task; o : TaskCompOp \\
Elt\_TaskSet\_TaskSet & \triangleq & ts_1, ts_2 : TaskSet; o : TaskCompOp; \\
 & & a : AnyAll \\
Set\_TaskSet\_TaskSet & \triangleq & ts_1, ts_2 : TaskSet; o : SetCompOp \\
SetCompOp & \triangleq & Identical \mid Subsetof \mid Overlap
\end{array}
$$

2.2 Sample Queries

In this section we will show some sample queries and how they can be captured in APQL in order to further illustrate the language. The sample queries, specified in natural language, are listed below (and numbered Q_1 to Q_{10}). In these queries, by default the value for the *AnyAll* identifier, when applicable, is *all*, and by default the value for the SimDegree is 1. Fig. 2 shows the grammar trees for queries Q_1 to Q_6, while Fig. 3 shows the grammar trees for queries Q_7 to Q_{10}. Note that in the following A to L are task labels (i.e. activity names).

Q_1. Select all process models where task A occurs in some process execution and task B occurs in every process execution;
Q_2. Select all process models where in every process execution task A may occur before task D;
Q_3. Select all process models where in every process execution task A always occurs before task D;
Q_4. Select all process models where in some process execution task A may occur before task B and task B may occur before task K;
Q_5. Select all process models where in some process execution task A always occurs before task B;
Q_6. Select all process models where task B occurs in parallel with task C;

Q_7. Select all process models where task B occurs in parallel with task C and where task A occurs in parallel with task H;

Q_8. Select all process models where in every process execution either task B occurs or task C;

Q_9. Select all process models where in every process execution the immediate predecessors of task H are among the immediate successors of task B;

Q_{10}. Select all process models where in some process execution the immediate predecessors of task H may occur after the common immediate successors of task B and task C;

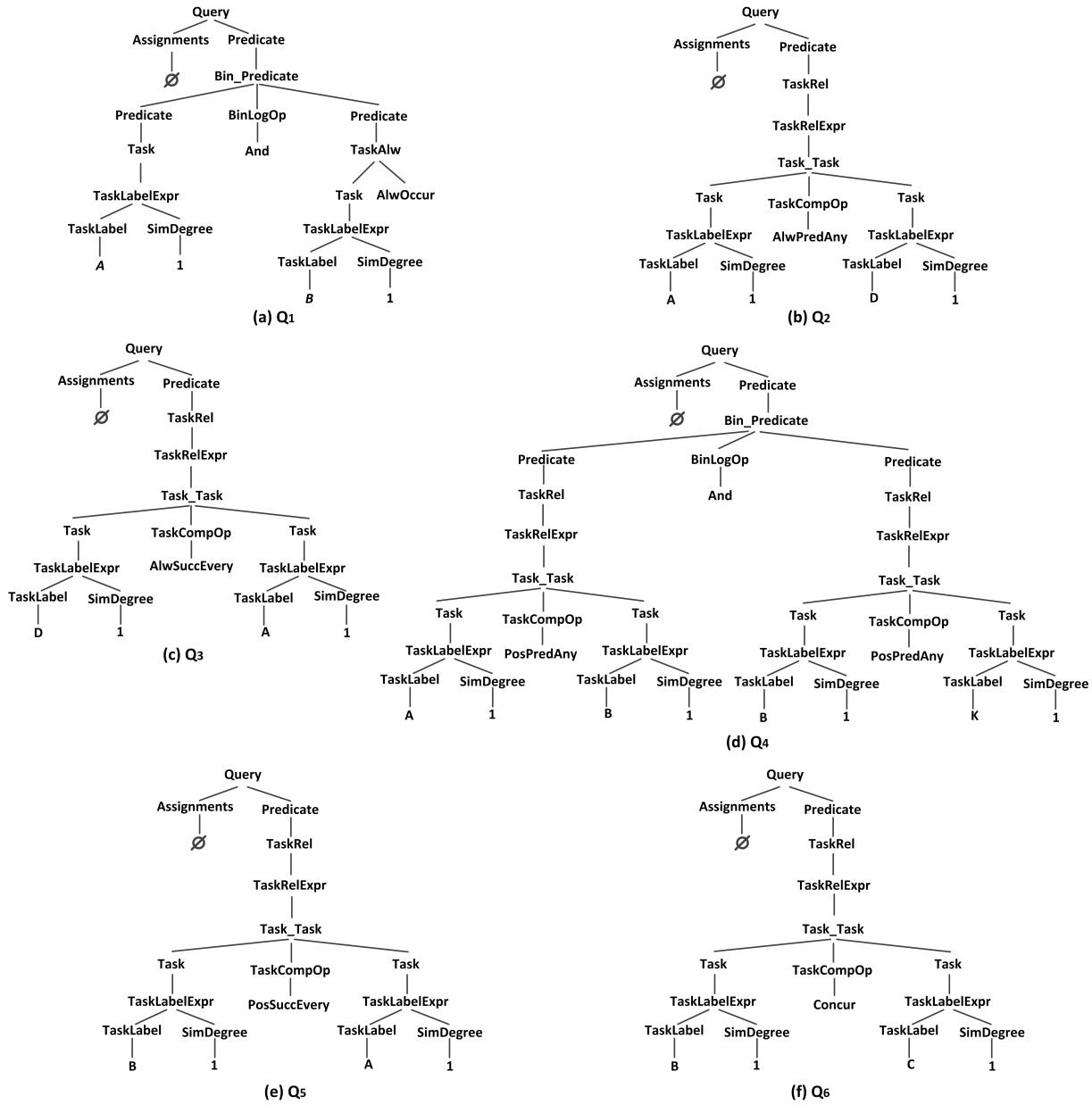

Fig. 2: The APQL grammar trees of sample queries $Q_1 - Q_6$

Fig. 3: The APQL grammar trees of sample queries $Q_7 - Q_{10}$

2.3 Semantics

In this section, we use denotational semantics to formally describe the semantics of APQL. For each nonterminal T we introduce a semantic function M_T which defines the meaning of the nonterminal in terms of its parts. The notation that we adopt throughout this section is the notation used in [7].

First, we introduce some auxiliary notation in order to facilitate the subsequent definition of the semantics.

Definition 1 (overriding union). *The overriding union of $f : X \to Y$ by $g : X \to Y$, denoted as $f \oplus g$, is defined by $g \cup f \backslash \{(x, f(x)) \mid x \in dom(f) \cap dom(g)\}$.*

With the set of 20 basic predicates defined in Section 2.1, we use $\mathbb{BP}_u$ to denote the set of two *unary predicates* $\{posoccur, alwoccur\}$ which specify unary task relations, and similarly we use $\mathbb{BP}_b$ to denote the set of other 18 *binary predicates* which specify binary task relations. The following two definitions introduce a higher order predicate that takes as input a unary or binary predicate, respectively. Note that the semantics of each predicate (ϕ/ψ) is language independent. For a task t in process model N, $L(t)$ specifies the label of t. A process model may have *slient tasks* which do not capture any task or activity in the process but are used for modelling purposes, e.g. a slient task used to capture an internal action that cannot observed by external uses. For a slient task t, we let $L(t) = \tau$.

Definition 2. *Let N be a process model and T the set of tasks in N, for $t_1, t_2 \in T$ and $\phi \in \mathbb{BP}_b$*

$$\mathsf{ref}^{\phi}(t_1, t_2, N) = \begin{cases} \phi(t_1, t_2, N) \textbf{ if } L(t_1) \neq \tau \wedge L(t_2) \neq \tau \\ FALSE \quad \textbf{otherwise} \end{cases}$$

i.e. the relation ϕ should hold between t_1 and t_2 in net N if both are non-silent tasks.

Definition 3. *Let N be a process model and T the set of tasks in N, for $t_1 \in T$ and $\psi \in \mathbb{BP}_u$*

$$\mathsf{ref}^{\psi}(t_1, N) = \begin{cases} \psi(t_1, N) \textbf{ if } L(t_1) \neq \tau \\ FALSE \quad \textbf{otherwise} \end{cases}$$

i.e. the relation ψ should hold for t_1 in net N if t_1 is a non-silent task.

As queries may use variables, we must know their values during query evaluation. A *Binding* is an assignment of task sets to variables:

$$Binding \triangleq ProcessModel \times Varname \rightarrowtail 2^{Task}$$

Queries are applied to a repository of process models, i.e.

$$Repository \triangleq 2^{ProcessModel}$$

A process model r consists of a collection of tasks T_r. For each task t in process model r we can retrieve its label as $L_r(t)$. Label similarity can be determined through the function Sim, where $Sim(l_1, l_2)$ determines the degree of similarity between labels l_1 and l_2 (which yields a value in the range [0,1]). Note that Sim is a parameter of the approach in which case one can choose his/her own similarity notion and returns the similarity evaluation result to this parameter.

The query evaluation function M_{Query} takes a query and a repository as input and yields the collection of process models in that repository that satisfy the query:

$$M_{Query} : Query \times Repository \to 2^{ProcessModel}$$

This function is defined as follows:

$$M_{Query}[q : Query, R : Repository] \triangleq M_{Predicate}(q.p, R, M_{Assignments}(q.s, R, \varnothing))$$

The evaluation of the query evaluation function depends on the evaluation of the predicate involved and the assignments involved. When evaluating a sequence of assignments we have to remember the values that have been assigned to the variables involved. Inititally this set of assignments is empty.

$$M_{Assignments} : Assignments \times Repository \times Binding \to Binding$$

The result of a sequence of assignments is a binding where the variables used in the assignments are bound to sets of tasks. If a variable was already assigned a set of tasks in an earlier assignment in the sequence the latest assignment takes precedence over the earlier assignment.

$M_{Assignments}[s : Assignments, R : Repository, B : Binding] \quad \triangleq$
if $\neg(s.TAIL).EMPTY$ **then**
$M_{Assignments}(s.TAIL, R, B \oplus M_{Assignment}(s.FIRST, R, B))$
else B

The result of an individual assignment is also a binding where the variable is linked to the set of tasks involved.

$M_{Assignment} : Assignment \times Repository \times Binding \rightarrow Binding$

$M_{Assignment}[a : Assignment, R : Repository, B : Binding)] \quad \triangleq$
$\{((r, a.v), M_{TaskSet}(a.ts, R, B)(r)) \mid r \in R\}$

A predicate can be evaluated in the context of a repository and a binding and the result is a set of process models from that repository.

$$M_{Predicate} : Predicate \times Repository \times Binding \rightarrow 2^{ProcessModel}$$

A predicate which is a task yields all process models in the repository that contain a task sufficiently similar to that task (with respect to the task label and similarity degree). A predicate which is a relationship between tasks (a *TaskRel* with a *TaskExpr*) yields all the process models that satisfy this relationship. A disjunction yields the union of the process models of the predicates involved, while a conjunction yields the intersection. The negation of a predicate yields the process models in the repository that do not satisfy the predicate.

$M_{Predicate}(p : Predicate, R : Repository, B : Binding) \quad \triangleq$
case p **of**
$Task \Rightarrow \{r \in R \mid \exists t \in T_r[Sim(p.l, L_r(t)) \geq p.d \wedge posoccur(t, r)]\}$
$TaskAlw \Rightarrow \{r \in R \mid \exists t \in T_r[Sim(p.l, L_r(t)) \geq p.d \wedge alwoccur(t, r)]$
$TaskRel \Rightarrow M_{TaskRel}(p.w, R, B)$
$Bin\_Predicate \Rightarrow$
case $p.o$ **of**
$And \Rightarrow M_{Predicate}(p.p_1, R, B) \cap M_{Predicate}(p.p_2, R, B)$
$Or \Rightarrow M_{Predicate}(p.p_1, R, B) \cup M_{Predicate}(p.p_2, R, B)$
end
$Un\_Predicate \Rightarrow R \backslash M_{Predicate}(p, R, B)$
end

A *TaskRel* with a *TaskRelExpr* in the context of a repository and a binding yields a set of process models in that repository.

$$M_{TaskRel} : TaskRelExpr \times Repository \times Binding \rightarrow 2^{ProcessModel}$$

A *TaskRelExpr* can be used to determine whether a task in a process model occurs in a given task set, whether a given basic predicate holds between a task in a process model and one or all tasks in a given task set, whether a given basic predicate holds between tasks in a process model, whether a given basic predicate holds between two or between all tasks in two given task sets, or whether a given set comparison relation holds between two given task sets.

$M_{TaskRel}(tr : TaskRelExpr, R : Repository, B : Binding) \quad \triangleq$
case tr **of**
$TaskInTaskSet \Rightarrow$
$\{r \in R \mid \exists v \in M_{TaskSet}(tr.ts, R, B)(r)[Sim(tr.t.l, L_r(v)) \geq tr.t.d]\}$
$Task\_TaskSet \Rightarrow$
case $tr.a$ **of**
$Any \Rightarrow \{r \in R \mid \exists t_1 \in T_r \; \exists t_2 \in M_{TaskSet}(tr.ts, R, B)(r)$
$[Sim(tr.t.l, L_r(t_1)) \geq tr.t.d \wedge rel^{tr.o}(t_1, t_2, r)]\}$
$All \Rightarrow \{r \in R \mid \exists t_1 \in T_r \; \forall t_2 \in M_{TaskSet}(tr.ts, R, B)(r)$

$[Sim(tr.t.l, L_r(t_1)) \geq tr.t.d \wedge rel^{tr.o}(t_1, t_2, r)]\}$

end

$Task\_Task \Rightarrow \{r \in R \mid \exists v_1, v_2 \in T_r[Sim(tr.t_1.l, L_r(v_1)) \geq tr.t_1.d \wedge$
$Sim(tr.t_2.l, L_r(v_2)) \geq tr.t_2.d \wedge rel^{tr.o}(v_1, v_2, r)]\}$

$Elt\_TaskSet\_TaskSet \Rightarrow$

case $tr.a$ **of**

$Any \Rightarrow \{r \in R \mid \exists t_1 \in M_{TaskSet}(tr.ts_1, R, B)(r)$
$\exists t_2 \in M_{TaskSet}(tr.ts_2, R, B)(r)[rel^{tr.o}(t_1, t_2, r)]\}$

$All \Rightarrow \{r \in R \mid \forall t_1 \in M_{TaskSet}(tr.ts_1, R, B)(r)$
$\forall t_2 \in M_{TaskSet}(tr.ts_2, R, B)(r)[rel^{tr.o}(t_1, t_2, r)]\}$

end

$Set\_TaskSet\_TaskSet \Rightarrow$

case $tr.o$ **of**

$Identical \Rightarrow$
$\{r \in R \mid M_{TaskSet}(tr.ts_1, R, B)(r) = M_{TaskSet}(tr.ts_2, R, B)(r)\}$

$Subsetof \Rightarrow$
$\{r \in R \mid M_{TaskSet}(tr.ts_1, R, B)(r) \subseteq M_{TaskSet}(tr.ts_2, R, B)(r)\}$

$Overlap \Rightarrow$
$\{r \in R \mid M_{TaskSet}(tr.ts_1, R, B)(r) \cap M_{TaskSet}(tr.ts_2, R, B)(r) \neq \varnothing\}$

end

end

A *TaskSet* within the context of a repository and a binding yields a mapping which assigns to each process model in the repository the collection of tasks within that model that satisfy the restriction imposed by the *TaskSet*.

$$M_{TaskSet} : TaskSet \times Repository \times Binding \rightarrow (ProcessModel \rightarrow 2^{Task})$$

When a *TaskSet* is a set of tasks, then for each process model the result is the set of tasks within that process model that are sufficiently similar to at least one of the tasks in that *TaskSet*. When the *TaskSet* is a variable, then the evaluation is similar except that the task set used is the task set currently bound to that variable. *TaskSets* can also be formed through *Construction* (where the set operators union, difference, and intersection are used) or *Application* (where task sets are formed through set comprehension, i.e. they are defined through properties that they have - these properties relate to the basic predicates).

$M_{TaskSet}(tks : TaskSet, R : Repository, B : Binding) \quad \triangleq$

case tks **of**

$SetofTasks \Rightarrow$
$\{(r, \{t \in T_r \mid \exists_{1 \leq i \leq tks.LENGTH}[Sim(tks(i).l, L_r(t)) \geq tks(i).d]\}) \mid r \in R\}$

$TaskSetVar \Rightarrow$
$\{(r, X) \mid r \in R\}$ **where**
$X = \begin{cases} B(r, tks.v) & \textbf{if } (r, tks.v) \in dom(B) \\ \varnothing & \textbf{otherwise} \end{cases}$

$Construction \Rightarrow$

case $tks.o$ **of**

$Union \Rightarrow$
$\{(r, M_{TaskSet}(tks.ts_1, R, B)(r) \cup M_{TaskSet}(tks.ts_2, R, B)(r)) \mid r \in R\}$

$Difference \Rightarrow$
$\{(r, M_{TaskSet}(tks.ts_1, R, B)(r) \backslash M_{TaskSet}(tks.ts_2, R, B)(r)) \mid r \in R\}$

$Intersection \Rightarrow$
$\{(r, M_{TaskSet}(tks.ts_1, R, B)(r) \cap M_{TaskSet}(tks.ts_2, R, B)(r)) \mid r \in R\}$

end

$Application \Rightarrow$

case $tks.a$ **of**

$Any \Rightarrow$
$\{(r, \{t \in T_r \mid \exists v \in M_{TaskSet}(tks.ts, R, B)(r)[rel^{tks.o}(t, v, r)]\}) \mid r \in R\}$

$All \Rightarrow$
$\{(r, \{t \in T_r \mid \forall v \in M_{TaskSet}(tks.ts, R, B)(r)[rel^{tks.o}(t, v, r)]\}) \mid r \in R\}$
end
end

2.4 Semantics of Sample Queries

In order to illustrate the formal semantics of APQL, a number of process models, represented in BPMN, are presented in Fig. 4. For each sample query of Section 2.2 and for each model it is indicated whether the model is part of the answer to the query (in that case the box corresponding to the query is ticked otherwise the box is not ticked). Note that in some models tasks with the same label occur (e.g. there are two tasks labeled A in model f).

3 Realisation in Petri nets

In this section, we demonstrate how the basic predicates can be derived for Petri nets. In doing so, APQL becomes a concrete query language for repositories of Petri net models. In order to be able to capture the computation of the basic predicates we introduce some basic Petri net concepts and terminology. In order to make the computation feasible, we also discuss the notion of unfolding. There are many papers discussing these concepts, and we refer the reader to [8] for an in-depth introduction to Petri net and to [9, 10, 5, 6] for unfolding and its related concepts.

3.1 Petri nets

Petri nets are a formal language of which the use for the specification of workflows has been argued by Wil van der Aalst (see e.g. [11, 12]). Petri nets can also be used as a formal foundation for defining the semantics of other process modelling languages or for reasoning about process models specified in these languages, for example BPMN [13], BPEL [14, 15], and EPCs [16].

Definition 4 (Petri nets). *A Petri net is a tuple (P, T, F), where:*

- *P is a finite set of places;*
- *T is a finite set of transitions, with $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$;*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, such that every transition is the source and the target of an arc.*

The conditions that the sets of places and transitions should be finite and that every transition has at least one input place and at least one output place derive from [6]. For notational convenience we adopt a commonly used notation, where $\bullet n$ represents all the inputs of a node n (which can be a place or a transition) and $n\bullet$ captures all its outputs.

Next, a labeled Petri net is basically a Petri net with annotated transitions and the annotation does not affect the semantics of the net.

Definition 5 (Labeled Petri nets). *A labeled Petri net is a tuple (P, T, F, A, L), where:*

- *(P, T, F) is a Petri net;*
- *A is a finite set of activity names;*
- *$L : T \rightarrow A \cup \{\tau\}$ is a labeling function for T, where $\tau \notin A$ is a silent action (i.e. an action not visible to the outside world).*

A *marking* of a Petri net is an assignment of tokens to its places. A marking represents a state of the net and a transition, if *enabled*, may change a marking into another marking, thus capturing a state change, by *firing*.

Definition 6 (Marking, Enabling and Firing of a Transition). *Let $PN = (P, T, F)$ be a Petri net.*

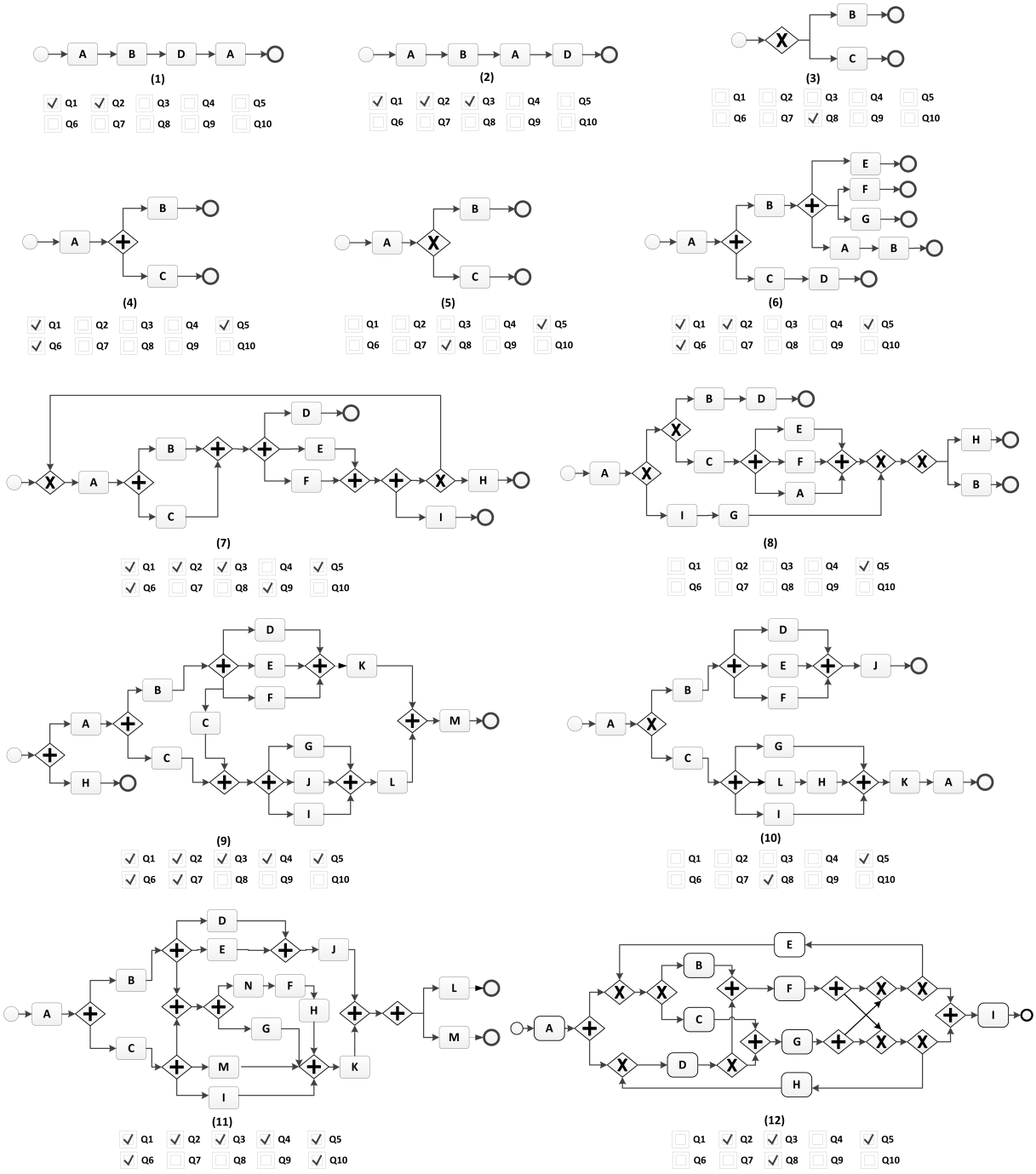

Fig. 4: Some BPMN business process models

- *A marking M of PN is a mapping $M : P \to \mathbb{N}$. A marking may be represented as a collection of pairs, e.g. $\{(p_0, 2), (p_1, 3), (p_2, 0)\}$ or as a vector, e.g. $2p_0 + 3p_1$ (in that case we drop places that do not have any tokens assigned to them). A* labeled Petri net system *is a labeled Petri net with an* initial marking *usually represented as M_0.*
- *Markings can be compared with each other, $M_1 \geq M_2$ iff for all $p \in P$, $M_1(p) \geq M_2(p)$. Similarly, one can define $>$, $<$, $\leq$, $=$.*
- *A transition t is* enabled *in a marking M, denoted as $M \xrightarrow{t}$, iff $M \geq \bullet t$.*
- *A transition t that is enabled in a marking M may* fire *and change marking M into M'. This is denoted as $M \xrightarrow{t} M'$.*

The markings of a Petri net system and the transition relation between these markings constitute a state space. In this paper we consider *n-bounded* Petri net systems (noting that such systems are always finite) which are necessary for the application of unfoldings.

Definition 7 (Reachability and Boundedness). *Let $\Sigma = (P, T, F, M_0)$ be a Petri net system.*

- *A marking M is called* reachable *if a transition sequence $\sigma = t_1 t_2 \dots t_n$ exists such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \dots \xrightarrow{t_n} M$, which may also be denoted as $M_0 \xrightarrow{\sigma} M$ or, if the choice of σ does not really matter, $M_0 \xrightarrow{*} M$.*
- *Σ is called a* finite *Petri net system if and only if its set of reachable markings is finite.*
- *Σ is called n-*bounded *iff for every reachable marking M and every place $p \in P$: $M(p) \leq n$.*

3.2 Unfolding

It is well-known that Petri nets may suffer from the *state space explosion* problem [17]. As such a naive exploration of the state space, especially in the context of a Petri net which allows highly concurrent behaviour, may not be tractable. In order to deal with this, McMillan [5] proposed a state space search technique based on the use of *unfolding* (this technique was later on improved by Esparza and Römer [6] and is discussed in the next sub-section). Unfoldings are applied to *n-bounded* (or called *n-safe* in [6]) Petri net systems and provide a method of searching the state space of concurrent systems without considering all possible interleavings of concurrent events. The concept of unfolding was firstly introduced by Nielsen et al. [9] and later elaborated upon by Engelfriet [10] using the term *branching processes*. Below we introduce the necessary concepts and notations to make this paper self-contained and to be able to build upon this theory. Most of these defintions are adopted from [6].

Firstly, various types of relationship may hold between pairs of nodes in a Petri net.

Definition 8 (Node relations (based on [6])). *Let $PN = (P, T, F)$ be a Petri net.*

- *F^+ is the irreflexive transitive closure of F, while F^* is its reflexive transitive closure. The partial orders defined by these closures are denoted as $<$ and $\leq$ respectively. Hence, for example, $x_1 < x_2$ iff $(x_1, x_2) \in F^+$ and we say that x_1* causally precedes *x_2.*
- *If $x_1 < x_2$ or $x_2 < x_1$ then x_1 and x_2 are* causally related*.*
- *Nodes x_1 and x_2 are in* conflict*, denoted by $x_1 \# x_2$, iff there exist distinct transitions $t_1, t_2 \in T$ such that $\bullet t_1 \cap \bullet t_2 \neq \varnothing$ and $t_1 \leq x_1$ and $t_2 \leq x_2$. A node x is in self-conflict iff $x \# x$.*
- *Nodes x_1 and x_2 are* concurrent*, denoted as x_1 co x_2, iff x_1 and x_2 are neither causally related nor in conflict.*

The unfolding of a Petri net is an *occurrence net*, usually infinite but with a simple, acyclic structure.

Definition 9 (Occurrence net (based on [6])). *An occurrence net is a net $N' = (B, E, F)$ where:*

- *B is a set of conditions;*
- *E is a set of events, with $B \cap E = \varnothing$;*
- *$F \subseteq B \times E \cup B \times E$ such that 1) for all $b \in B$, $|\bullet b| \leq 1$, 2) F is acyclic, i.e. F^+ is a strict partial order, and 3) for all $x \in B \cup E$ the set of nodes $y \in B \cup E$ for which $y < x$ is finite;*
- *No node is in self-conflict, i.e. for all $x \in B \cup E$, $\neg(x \# x)$.*

We also adopt the notion of $\mathsf{Min}(N')$, as in [6], to denote the set of minimal elements of N' with respect to the strict partial order F^+. As for transitions in Petri nets, we only consider events that have at least one input and at least one output condition. The minimal elements are therefore conditions only, and intuitively $\mathsf{Min}(N')$ can be seen as an initial marking of the net.

Definition 10 (Branching process (based on [10])). *A branching process of a Petri net system $\Sigma = (N, M_0)$, with $N = (P, T, F)$, is a pair (N', h), where*

- *$N' = (B, E, F)$ is an occurrence net;*
- *$h : N' \to N$ is a homomorphism which, following [10], means that:*
  - *$h(B \cup E) \to (P \cup T)$;*
  - *$h \subseteq (B \times P) \cup (E \times T)$, i.e. conditions are mapped to places and events to transitions;*
  - *For every $t \in T$, $h[\bullet t]$ is a bijection between $\bullet t$ and $\bullet h(t)$, and $h[t\bullet]$ is a bijection between $t\bullet$ and $h(t)\bullet$;*
  - *$h[\mathsf{Min}(N')]$ is a bijection between $\mathsf{Min}(N')$ and $\{p \in P \mid M_0(p) > 0\}$.*
- *For all $e, e' \in E$, if $h(e) = h(e')$ and $\bullet e = \bullet e'$, then $e = e'$.*

Note that the definition allows for infinite branching processes. In [10] it is shown that, up to isomorphism, every net system has a unique maximal branching process. For a net system Σ, this unique process is referred to as the *unfolding* of Σ and it is denoted as Unf_Σ. For example, in Fig. 5 the Petri nets in (a) can be unfolded into the occurrence net in (b). Note that in Fig. 5(b) all (condition/event) nodes are identified by integers and annotated by the corresponding place or transition identifiers in Fig. 5(a).

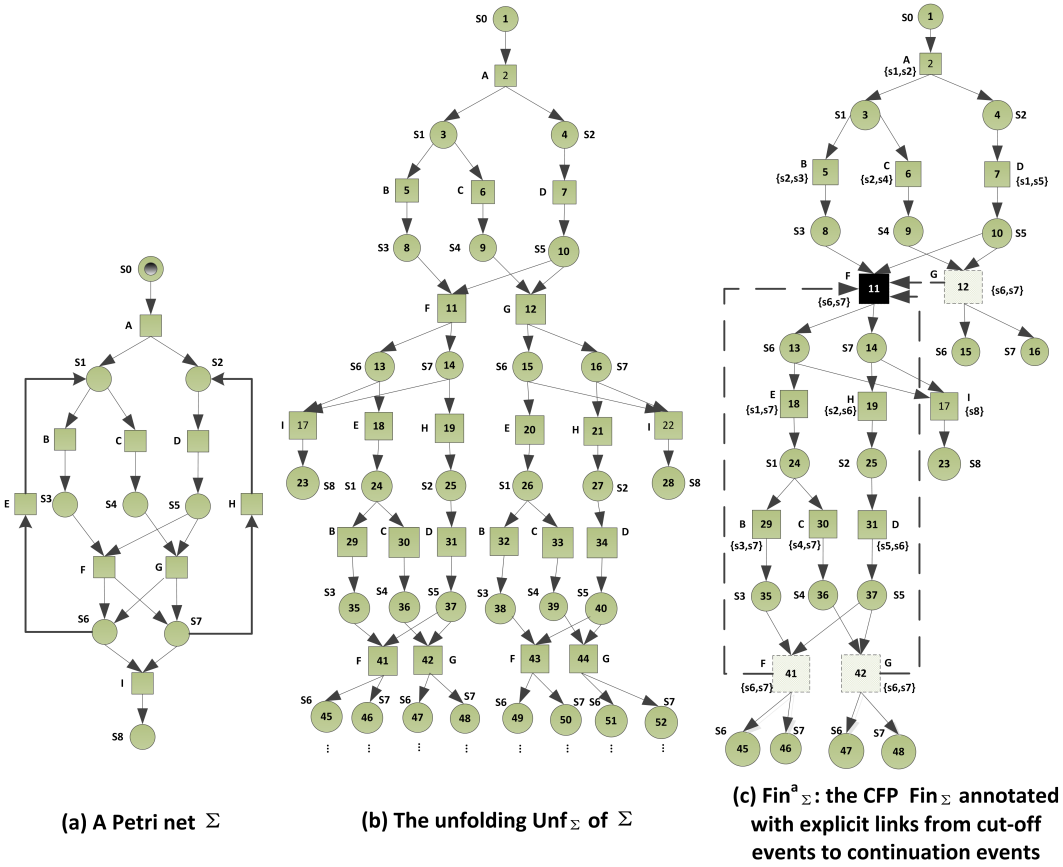

Fig. 5: Illustration of unfolding and complete finite prefix using the Petri net Σ of the BPMN model in Fig. 4(12) (the net in (a) without s_0, s_8, A and I is the same as the example net in Fig. 1 in [6]).

3.3 Complete Finite Prefix

The unfolding of a Petri net is infinite when the net is cyclic, as for example Unf_Σ in Fig. 5(b). In [5], McMillan proposed an algorithm for the construction of a so-called truncated unfolding, which is a finite initial part of an unfolding and contains as much reachability information as the unfolding itself but may be much larger than necessary. In [6], Ezparza and Römer referred to this truncated unfolding as *complete finite prefix* (CFP) and proposed an improved algorithm for computing a minimal CFP. For example, as illustrated in Fig. 5(c) (the dashed arcs should be ignored for the moment), Fin_Σ is a minimal CFP of Σ. Note that in Fig. 5(c) the tuple of conditions positioned next to an event node represents the marking of the net upon the occurrence of that event.

The main theoretical notions required to understand the concepts of a CFP are that of *configuration* and *local configuration* of events. Firstly, a configuration represents a possible partially ordered run of the net.

Definition 11 (Configuration [6]). *A configuration C of an occurrence net $N = (B, E, F)$ is a set of events, i.e. $C \subseteq E$, satisfying the following two conditions:*

- *C is causally downward-closed, i.e. $(e \in C \wedge e' \leq e) \Rightarrow e' \in C$*
- *C is conflict free, i.e. for all $e, e' \in C : \neg(e \# e')$*

Given a configuration C the set of places Cut_C represents a reachable marking, which is denoted by $Mark(C)$. In other word, $Mark(C)$ is the marking to reach by firing the configuration C. For example, in the unfolding Unf_Σ in Fig. 5(b) we have $Mark(\{2, 5, 7, 11, 17\}) = \{s_8\}$.

Definition 12 (Cut [6]). *Let Σ be a Petri net system and (N', h) be its unfolding. The set of conditions associated with a configuration of N' is called a* cut*, and is defined as $Cut_C = (\mathsf{Min}(N') \cup C\bullet) \setminus \bullet C$. A cut uniquely defines a reachable marking in Σ: $Mark(C) = h(Cut_C)$.*

The concepts thus far can be used to introduce the unfolding algorithm. In [6] a branching process (N', h) of a Petri net system Σ is specified as a collection of nodes. These nodes are either conditions or events. A condition is a pair (s, e) where e is the input event of s, while an event is a pair (t, B) where t is a transition and B its input conditions. A set of conditions of a branching process is a *co-set* if its elements are pairwise in co relation. For example, in Fig. 5(b) each of the node sets {13,14}, {15,16}, {45,46}, {47,48}, {49,50} and {51,52} is a co-set.

During the process of unfolding the collection of nodes increases where the function $PE(N', h)$ (which denotes the possible extensions) is applied to determine the nodes to be added. The possible extensions are given in the form of event pairs (t, B), where B is a co-set of conditions of (N', h) and t is a transition of Σ such that 1) $h(B) = \bullet t$ and 2) no event e exists for which $h(e) = t$ and $\bullet e = B$. In the unfolding algorithm, nodes from the set of possible extensions $PE(N', h)$ are added to the unfolding of the net till this set is empty (i.e. there are no more extensions).

In the complete finite prefix approach, it is observed that a finite prefix of an unfolding may contain all reachability-related information. The key to obtaining a CFP is to identify those events at which we can cease unfolding (e.g. events 12, 41 and 42 in Fin_Σ in Fig. 5(c)) without loss of reachability information. Such events are referred to as *cut-off events* and they are defined in terms of an *adequate order* on configurations.

Definition 13 (Adequate order [6]). *Let $\Sigma = (P, T, F, M_0)$ be a Petri net system and let $\prec$ be a partial order on the finite configurations of one of its branching processses, then $\prec$ is an adequate order iff:*

- *$\prec$ is well founded;*
- *For all configurations C_1 and C_2, $C_1 \subset C_2 \Rightarrow C_1 \prec C_2$;*
- *The $\prec$ order is preserved in the context of finite extensions, i.e. if $C_1 \prec C_2$ and $Mark(C_1) = Mark(C_2)$, then if we extend C_1 with E to C_1' and we extend C_2 to C_2' by using an extension isomorphic to E then $C_1' \prec C_2'$.*

The last clause of this definition is not fully formalised here as it requires a certain amount of formalism and we hope that the idea is sufficiently clear from an intuitive point of view. We refer the reader to [6] for a complete formal definition of this notion. Note that, as pointed out in [6], the order $\prec$ is essentially a parameter to the approach.

The concept of *local configuration* captures the idea of all preceding events to an event such that these events form a configuration.

Definition 14 (Local Configuration [6]). *Let $N = (B, E, F)$ be an occurrence net, the local configuration of an event $e \in E$, denoted $[e]$, is the set of events e', where $e' \in E$, such that $e' \leq e$.*

Definition 15 (Cut-off event [6]). *Let Σ be a Petri net system, N' be one of its branching processes and let $\prec$ be an adequate order on the configurations of N', then an event e is a cut-off event iff N' contains a local configuration $[e']$ for which $Mark([e]) = Mark([e'])$ and $[e'] \prec [e]$.*

Without loss of reachability information, we can cease unfolding from an event e, if e takes the net to a marking which can be caused by some earlier other event e'. So in Fig. 5(c), we remove the part after event 12 from Unf_Σ because it is isomorphic to that after event 11, i.e. the continuation after event 12 is essentially the same as the continuation after event 11. For a proof of this approach we refer to [6].

3.4 Annotating Complete Finite Prefix

In this work, the repository of process models are captured in terms of CFPs. All (binary) predicates between tasks are determined by examing the possible firing sequences in the CFP of each process model. To facilitate our algorithms for determing these predicates (presented in the next sub-section), we would like to represent the continuation from cut-off events slightly more explicit in a CFP. The idea is that for each of the cut-off events e in a CFP we mark out some earlier other event e' that can lead to the same marking as e (i.e. $Mark([e]) = Mark([e'])$ and $[e'] \prec [e]$). We referred to e' as the *continuation event* of e in the CFP. We then annotate the CFP with links that connect from each cut-off event to its continutation event.

Definition 16 (Notations of continuation events and cut-off events). *Let $\Sigma = (N, M_0)$ be a Petri net system, with $N = (P, T, F)$, and let $\rho = (N', h)$, with $N' = (B', E', F')$, be an unfolding of Σ, then we define:*

- $\mathsf{Eq}(M, \rho) = \{e \in E' \mid Mark([e]) = M\}$ *for any reachable marking M of N. If ρ is clear from the context, we will simply omit it and write $\mathsf{Eq}(M)$ (a similar convention holds for the remainder of this definition and ρ is not introduced explicitly anymore).*
- $\mathsf{continuation}(M)$ *which refers to the continuation node in ρ for a reachable marking M. It is defined as the unique event $e' \in \mathsf{Eq}(M)$ such that for all $e \in \mathsf{Eq}(M)$, if $e \neq e'$ then $[e'] \prec [e]$.*
- $\mathsf{cutoff}(M) = \mathsf{Eq}(M) \backslash \{\mathsf{continuation}(M)\}$ *which denotes the set of cut-off events for a reachable marking M.*

Definition 17 (Annotated Complete Finite Prefix). *Let $\Sigma = (N, M_0)$ be a Petri net system, Fin^a_Σ denotes a CFP of Σ that is annotated with links from cut-off events to their continuation events, shortly referred to as an* annotated CFP. *$\mathsf{Fin}^a_\Sigma = (\mathsf{Fin}_\Sigma, L)$ where:*

- *$\mathsf{Fin}_\Sigma = (B, E, G)$ is the CFP of Σ;*
- *L is a set of links defined as $L \subseteq E \times E$, and if and only if $(e, e') \in L$, then there is a reachable marking M such that $e' = \mathsf{continuation}(M)$ and $e \in \mathsf{cutoff}(M)$.*

Example 1 *Consider Fin^a_Σ as shown in Fig. 5(c). For this annotated CFP, $L = \{(41, 11), (42, 11), (12, 11)\}$.*

To generate an annotated CFP, we propose a slight adaptation of the algorithm for computing a CFP for a n-safe net system in [6]. This adapted algorithm is presented as Algorithm 1. Based on Definition 17, the data structure for the representation of an annotated CFP comprises that of a CFP in [6] (written $\mathsf{Fin}^a.N$) and a set of links (written $\mathsf{Fin}^a.L$). $PE(\mathsf{Fin}^a.N)$ is the set of events that can be added to a branching process $\mathsf{Fin}^a.N$ (i.e. possible extensions of $\mathsf{Fin}^a.N$), as defined in [6]. Application of $minimal(pe, \prec)$ yields an event e which satisfies the following condition taken from [6]: $e \in pe$ and $[e]$ is minimal with respect to $\prec$. The predicate $expansion\_required(e, cut\_off)$ is an abbreviation of $[e] \cap cut\_off = \varnothing$, the condition used in [6]. Next, $cut\_off\_event(e, \mathsf{Fin}^a.N, c)$ returns the result of whether or not e is a cut-off event of $\mathsf{Fin}^a.N$ (as in [6]), and during its application, the corresonding continuation event for e is returned in the local variable c so that it does not need to be determined again when adding links. Note that we use $X \cup\!:= Y$ as an abbreviation for $X := X \cup Y$, and $X \setminus\!:= Y$ for $X := X \setminus Y$.

Algorithm 1: Computation of an annotated CFP via an adaption of Algorithm 4.7 in [6]

Input: An n-safe Petri net system $\Sigma = (P,\ T,\ F,\ M_0)$
Output: $\mathsf{Fin}^a\ (N : Net, L : Links)$ an annotated CFP of Σ
begin
  $\mathsf{Fin}^a.N := \{(s, \varnothing) \mid M_0(s) > 0 \wedge s \in P\}$;
  $\mathsf{Fin}^a.L := \varnothing$;
  $pe := PE(\mathsf{Fin}^a.N)$;
  $cut\_off := \varnothing$;
  while $pe \neq \varnothing$ **do**
    $e := minimal(pe, \prec)$;
    if $expansion\_required(e, cut\_off)$ **then**
      $\mathsf{Fin}^a.N \cup := \{(e.t, \bullet e)\} \cup \{(s, e) \mid s \in e.t\bullet\}$;
      $pe := PE(\mathsf{Fin}^a.N)$;
      if $cut\_off\_event(e, \mathsf{Fin}^a.N, c)$ **then**
        $cut\_off \cup := \{e\}$;
        $\mathsf{Fin}^a.L \cup := \{(e, c)\}$;
    else $pe \setminus := \{e\}$

3.5 Determining the Basic Predicates

In Section 2, we defined a set of 20 basic predicates based on process execution semantics, and to check if such a predicate holds requires in principle exploration of *all* process executions. Since different process executions result from choices in a process model, we propose to pre-process the annotated CFP of each process model (Algorithm 2) as follows: first we transform such a CFP to a set of conflict-free CFPs (specified by function *GetAllExecutions* in Algorithms 3) and then convert each resulting CFP to a directed bipartite graph (or bigraph) (specified by *AnnotatedCFP2Bigraph* in Algorithm 5).

Algorithm 2: Preprocessing an annotated CFP to a set of directed bigraphs

function *PreProcess*(U: annotated CFP): set of bigraphs $\mathbb{G}$
begin
  $\mathbb{G} := \varnothing$;
  $\mathbb{U}$:= *GetAllExecutions*(U);
  for $U \in \mathbb{U}$ **do**
    $\mathbb{G} \cup :=$ *AnnotatedCFP2Bigraph*(U)

In Algorithm 3, GetLeafCondCoSets yields all co-sets of leaf conditions in the input CFP. By traversing backwards the input CFP (without considering the set of links) from each of these co-sets, ComputeCFPs produces the set of CFPs as a decomposition of the input CFP. This set of CFPs are free of conflicts due to the co relation between the leaf conditions in each co-set. For illustration, Fig. 6 depicts the set of conflict-free CFPs as decomposition of Fin_Σ in Fig. 5(c) via computation of GetLeafCondCoSets and ComputeCFPs.

Next, we convert the link annotations of the input CFP to the link annotations for each of the conflict-free CFPs (that result from the above decomposition of the input CFP). If such a CFP does not contain a cut-off event ($E_{cutoff} = \varnothing$), there is no link annotation and the CFP will remain as it is. Otherwise, for a CFP with cut-off events, there are two cases to consider depending on whether a cut-off event (e_{cut}) in the CFP links to a continuation event (e_{cont}) within or outside this CFP. If the CFP contains both events, the link (e_{cut}, e_{cont}) is directly added into the link annotations of the CFP. Otherwise, if the CFP contains e_{cut} but not e_{cont}, we propose to update the CFP (specified by function GetUpdatedCFPs in Algorithm 4) and the link annotations till there exists no link across two different CFPs.

Algorithm 4 specifies how to update a CFP with a cut-off event linking to a continuation event outside the CFP. The basic idea is to identify among the set of conflict-free CFPs (Γ) those (ρ_i) that contain e_{cont},

and to replace the part before and including e_{cont} in such a CFP (ρ_i) with the part before and including e_{cut} in the original CFP (ρ). This results in the same number of updated CFPs (ρ') as that of the CFPs containing e_{cont}. Since e_{cont} is replaced by e_{cut} in the updated CFPs and (e_{cut}, e_{cont}) is not used any more, the link annoations need update as well.

Back to Algorithm 3, we retrieve the links (L_{add}) that lead to e_{cont} except for (e_{cut}, e_{cont}), and replace e_{cont} with e_{cut} in these links. Accordingly, the flag f_{update} is set to TRUE signaling the fact that CFP updates have been applied, and the updated CFPs are added to the set of remaining CFPs (Γ_{tmp}) for processing of link annotations. For a given CFP (ρ'), if all the cut-off events in the CFP are processed without CFP updates ($\neg f_{update}$), the set of links (L') that are computed from such processing are added as the CFP's link annotations. The above procedure for converting link annotations is repeated till there are no more remaining CFPs ($\Gamma_{tmp} = \varnothing$). For illustration, Fig. 7 depicts the set of conflict-free annotated CFPs as decomposition of Fin^a_Σ in Fig. 5(c) via computation of Algorithm 3. Note that Fig. 7(d)-(f) show the three updated CFPs as result of combining the part before and including cut-off event 12 in the CFP in Fig. 6 (d) with the part

Algorithm 3: Transforming an annotated CFP into a set of conflict-free annotated CFPs

function *GetAllExecutions*
Input: An annotated CFP $U = (\rho, L)$ where $\rho = (B, E, F)$ and $L \subseteq E \times E$
Output: A set of annotated CFPs $\mathbb{U}$
begin
  $\mathbb{U} := \varnothing$;
  $\Gamma := \varnothing$;
  `/* compute CFPs from each of the co-sets of leaf conditions */`
  CS := GetLeafCondCoSets(ρ);
  for $cs \in CS$ **do**
    $\Gamma \cup:= \{\text{ComputeCFP}(\rho, cs)\}$;
  `/* generate annotated CFPs from the above (conflict-free) CFPs */`
  $\Gamma_{tmp} := \Gamma$;
  repeat
    Select $\rho' \in \Gamma_{tmp}$;
    $L' := \varnothing$;
    E_{cutoff} := GetCutoffEvents(ρ');
    f_{update} := FALSE; `/* the flag changes to TRUE if there are CFP updates */`
    while $E_{cutoff} \neq \varnothing \wedge \neg f_{update}$ **do**
      Select $e_{cut} \in E_{cutoff}$;
      e_{cont} := GetContinuationEvent(L, e_{cut});
      if $e_{cont} \in \rho'.E$ **then**
        $L' \cup:= \{(e_{cut}, e_{cont})\}$;
      else
        Γ_{add} := *GetUpdatedCFPs*($\rho', \Gamma, e_{cut}, e_{cont}$); `/* see Algorithm 4 */`
        $\Gamma \setminus:= \{\rho'\}$;
        $\Gamma \cup:= \Gamma_{add}$;
        L_{add} := GetLinks_to(L, e_{cont}) $\setminus \{(e_{cut}, e_{cont})\}$;
        for *e where* $(e, e_{cont}) \in L'$ **do**
          $L_{add} \cup:= \{(e, e_{cut})\}$
        $L \setminus:= \{(e_{cut}, e_{cont})\}$;
        $L \cup:= L_{add}$;
        f_{update} := TRUE; `/* set the flag to TRUE upon CFP updates */`
        $\Gamma_{tmp} \cup:= \Gamma_{add}$; `/* add to the remaining CFPs for link annotations */`
      $E_{cutoff} \setminus:= \{e_{cut}\}$;
    if $\neg f_{update}$ **then**
      $\mathbb{U} \cup:= \{(\rho', L')\}$;
    $\Gamma_{tmp} \setminus:= \{\rho'\}$
  until $\Gamma_{tmp} = \varnothing$;

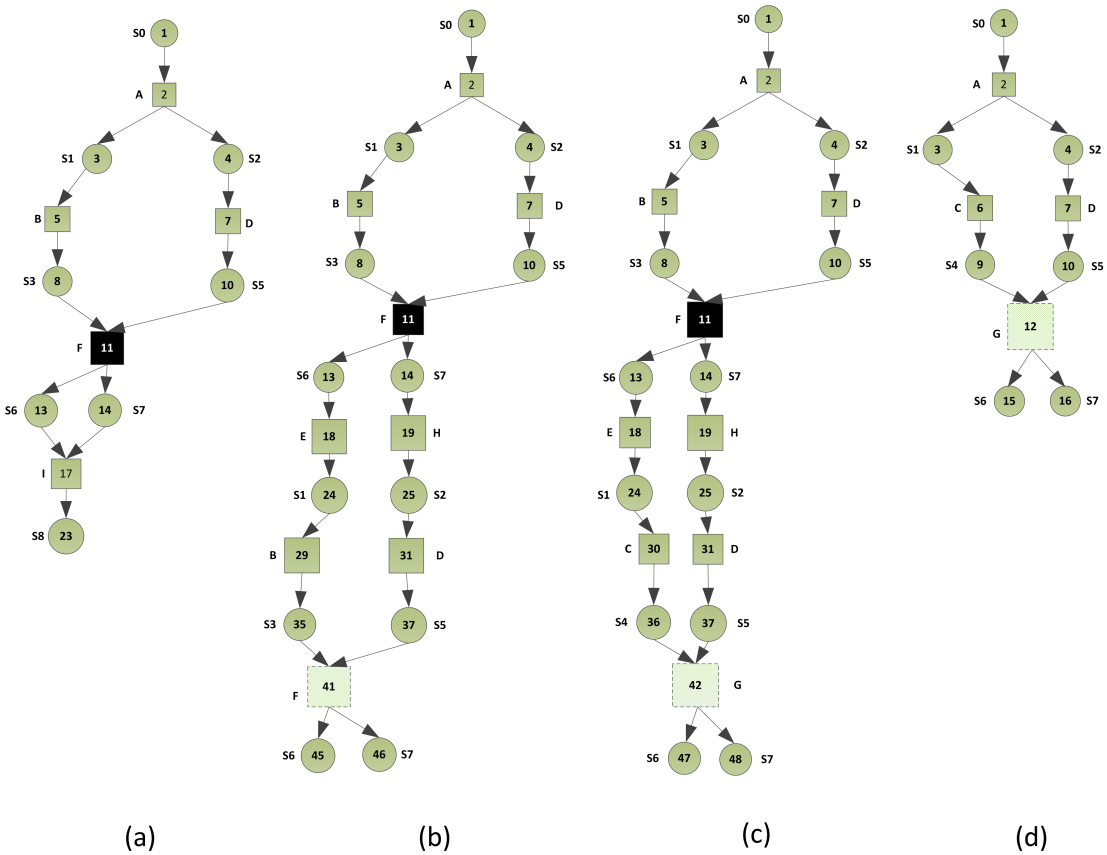

Fig. 6: The set of conflict-free CFPs as decomposition of Fin_{Σ} in Fig. 5(c).

Algorithm 4: Updating a CFP with a cut-off event that links to a continuation event outside the CFP

function *GetUpdatedCFPs*
Input: A CFP $\rho = (B, E, F)$, a set of CFPs Γ, a (cut-off) event e_{cut}, a (continuation) event e_{cont}
Output: A set of (updated) CFPs Γ'
begin
  $\Gamma' := \varnothing$;
  `/* get` ρ `ready by removing the successor conditions of` e_{cut} `(in` ρ`) */`
  B_{tmp} := iSuccessors(ρ, e_{cut});
  $\rho.B \setminus := B_{tmp}$;
  $\rho.F \setminus := \{e_{cut}\} \times B_{tmp}$;
  `/* retrieve and process the CFPs that contain` e_{cont} `in` Γ `*/`
  for $\rho_i \in \Gamma$ *where* $e_{cont} \in \rho_i.E$ **do**
    `/* remove from` ρ `the part before` e_{cont}`,` e_{cont} `itself, and the outgoing edges of` e_{cont} `*/`
    H := GetSubCFP_to(ρ_i, e_{cont});
    $\rho_i.B \setminus := H.B$;
    $\rho_i.E \setminus := H.E$;
    $\rho_i.F \setminus := H.F \cup (\{e_{cont}\} \times$ iSuccessors$(\rho_i, e_{cont}))$;
    `/* connect the above (updated)` ρ `and` ρ_i `to` ρ' `*/`
    $\rho'.B$:= $\rho.B \cup \rho_i.B$;
    $\rho'.E$:= $\rho.E \cup \rho_i.E$;
    $\rho'.F$:= $\rho.F \cup \rho_i.F \cup (\{e_{cut}\} \times$ InitialConditions$(\rho_i))$;
    $\Gamma' \cup := \{\rho'\}$

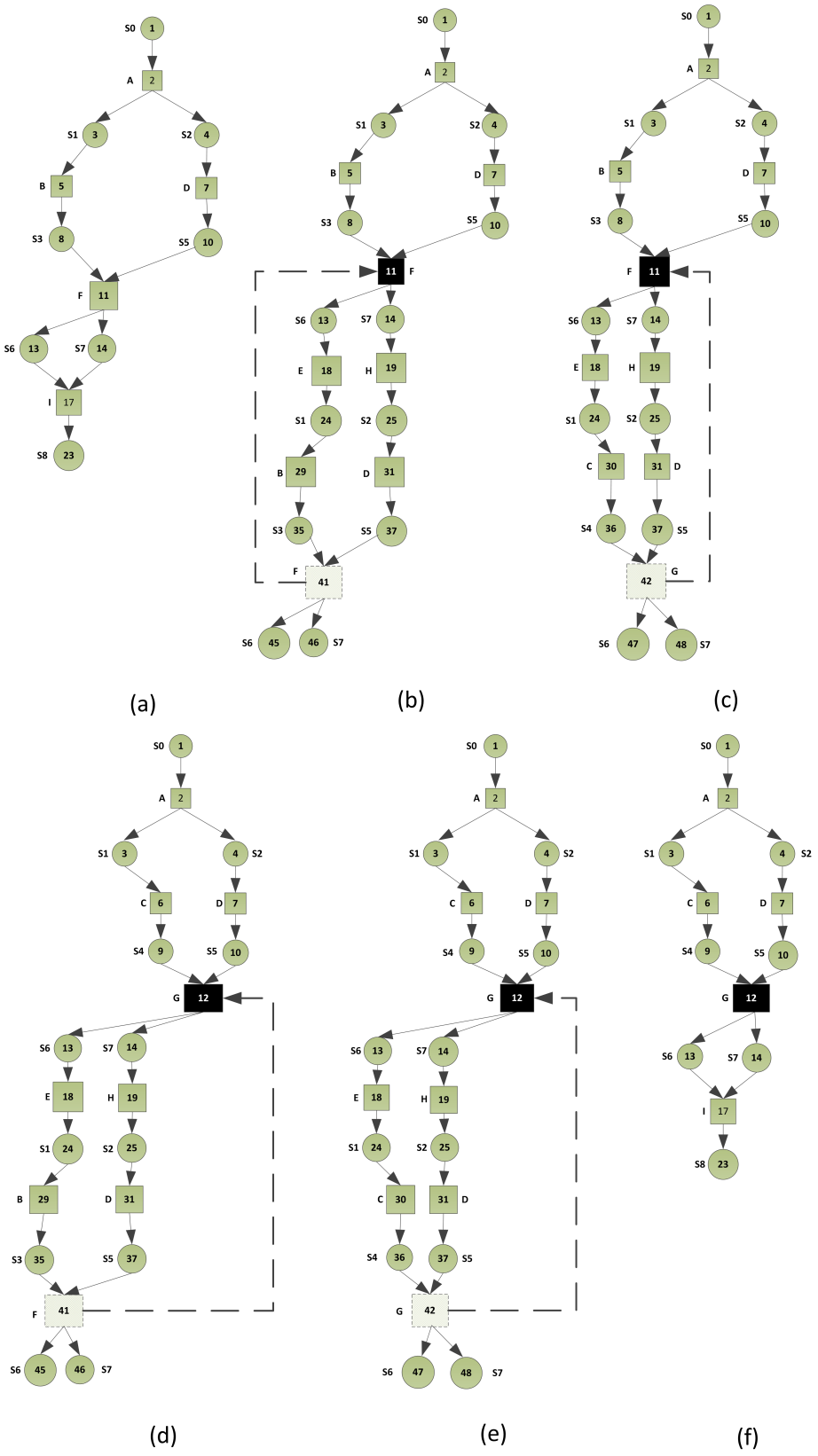

Fig. 7: The set of conflict-free annotated CFPs tranformed from $\mathsf{Fin}^{a}_{\Sigma}$ in Fig. 5(c) according to Algorithm 3.

after continuation event 11 in each of the CFPs in Fig. 6(a)-(c), respectively, and then replacing continuation event 11 with event 12 in the corresponding CFPs.

Finally, Algorithm 5 specifies how to convert an annotated CFP into a directed bigraph. The transformation is straight-forward where the events in the CFP become event nodes in the bigraph, conditions become condition nodes, the arcs become the directed edges, and the links are converted to the edges leading from a cut-off event to each of the immediate successors (conditions) of the corresponding continuation event. For illustration, Fig. 8 depicts an example of converting an annotated CFP to a directed bigraph.

Algorithm 5: Converting an annotated CFP to a directed bigraph

function *AnnotatedCFP2Bigraph*
Input: An annotated CFP $U = (\rho, L)$ where $\rho = (B, E, F)$ and $L \subseteq E \times E$
Output: A directed bigraph $G = (V_{cond}$: condition nodes, V_{event}: event nodes, A: directed edges)
begin
  $V_{cond} := B$;
  $V_{event} := E$;
  $A := F$;
  for $(e_1, e_2) \in L$ **do**
    $B_1 := \mathsf{iSuccessors}(e_1)$;
    $B_2 := \mathsf{iSuccessors}(e_2)$;
    $V_{cond} \setminus := B_1$;
    $A \setminus := \{e_1\} \times B_1$;
    $A \cup := \{e_1\} \times B_2$;

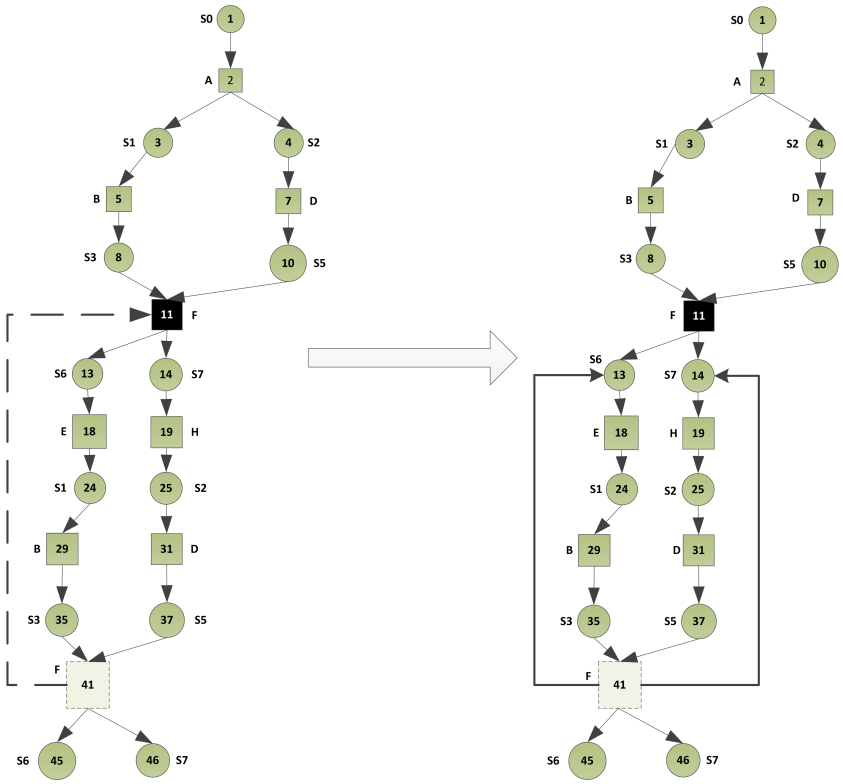

Fig. 8: Converting a conflict-free annotated CFP to a directed bigraph

Now we define the algorithms for determining the 20 basic predicates. These algorithms are performed to provide input for evaluation of $M_{Predicate}$ in Section 2.3. More specifically, they produce results of $posoccur(t, r)$, $alwoccur(t, r)$, and $rel^{tr.o}(t_1, t_2, r)$ in $M_{TaskRel}$ (where $tr.o$ specifies one of the 18 binary predicates). Also, we introduce two common functions: RetrieveBigraphs which returns the set of bigraphs for a process model (r) from the above pre-processing, and RetrieveAllEvents which returns the set of event nodes for (i.e. labeled with) a task (t) in a bigraph (G). Each such bigraph represents a possible execution of the corresponding process, and each event node labeled with a task identifier in the bigraph captures an occurrence of the corresponding task in that process execution. For a short notation, an event node labeled with task t is hereafter referred to as an t-event node.

Algorithms 6 and 7 specifies how to evaluate the two unary predicates. Predicate $posoccur$ or $alwoccur$ of task t in process model r can be determined by checking the presence of a t-event node in *any* or *all* bigraphs of r. Based on the fact that the set of bigraphs of process model r are each free of choices, the $exclusive$ relation between two tasks t and t' is determined by checking in *every* bigraph of r if there are both a t-event node and a t'-event node, as specified in Algorithm 8. In Algorithm 9, the $concur$ relation between t and t' in r holds if and only if in each bigraph of r either 1) there are no t- and t'-event nodes at all, or 2) there are both an t-event node and an t'-event node and no directed path exists between the two nodes.

Next, the remaining algorithms are defined for predicates capturing causal relationships between tasks. Evaluation of each such predicate is based on the result of evaluating the corresponding intermediate predicate in individual process executions. There are *Alw-* predicates and *Pos-* predicates. Given a process model r, an *Alw-* predicate (e.g. *Alw*SuccEvery) holds only when its intermediate predicate (e.g. SuccEvery) holds in *all*

Algorithm 6: Determining the (unary) predicate PosOccur

function POSOCCUR(t: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ (RetrieveAllEvents(G, t) $\neq \varnothing$)

Algorithm 7: Determining the (unary) predicate AlwOccur

function ALWOCCUR(t: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ (RetrieveAllEvents(G, t) $\neq \varnothing$)

Algorithm 8: Determining the predicate Exclusive

function EXCLUSIVE(t: taskID, t': taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ (RetrieveAllEvents(G, t) $\cap$ RetrieveAllEvents(G, t') $= \varnothing$)

Algorithm 9: Determining the predicate Concur

function CONCUR(t: taskID, t': taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ ((RetrieveAllEvents(G, t) $= \varnothing \wedge$ RetrieveAllEvents(G, t') $= \varnothing$) $\vee$
    (RetrieveAllEvents(G, t) $\neq \varnothing \wedge$ RetrieveAllEvents(G, t') $\neq \varnothing \wedge$
      $\forall_{e \in \text{RetrieveAllEvents}(G,t)} \forall_{e' \in \text{RetrieveAllEvents}(G,t')}$ [NoDirectedPath(e, e', G) $\wedge$ NoDirectedPath(e', e, G)]))

process executions of r, while a *Pos*- predicate (e.g. *Pos*SuccAny) holds as long as its intermediate predicate (e.g. SuccAny) holds in *one* process execution of r. To capture such semantics, we apply logical operator $\bigwedge$ (for an *Alw*- predicate) or $\bigvee$ (for a *Pos*- predicate) between the corresponding intermediate predicate over the set of bigraphs ($\mathbb{G}$) of r in the algorithms. For example, Algorithm 10 specifies the evaluation of predicate AlwSuccEvery and Algorithm 10 the evaluation of PosSuccAny.

Algorithm 10: Determining the predicate AlwSuccEvery

function ALWSUCCEVERY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G} :=$ RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *SuccEvery*$(t_{former}, t_{latter}, G)$

Algorithm 11: Determining the predicate PosSuccAny

function POSSUCCANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G} :=$ RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *SuccAny*$(t_{former}, t_{latter}, G)$

Let us move on to the algorithms for evaluation of intermediate predicates. There are eight intermediate predicates resulting from combinations of *-Every* vs. *-Any* predicates and root predicates for causal relationship (i.e. *Succ*, *ISucc*, *Pred* and *IPred*). Consider an execution i of process model r and two tasks t_1 and t_2 in r. An *-Every* predicate over t_1 and t_2 holds only when *every* occurrence of t_1 is causally related to (i.e. succeeded or preceded by) an occurrence of t_2, while an *-Any* predicate over t_1 and t_2 holds as long as *one* occurrence of t_1 is causally related to an occurrence of t_2. To capture such semantics, we apply logical operator $\forall$ (for an *-Every* predicate) or $\exists$ (for an *-Any* predicate) over the set of instances of t_1 in the algorithms. For example, Algorithm 12 specifies the evaluation of intermediate predicate SuccEvery and Algorithm 13 the evaluation of SuccAny. In these two algorithms, t_{former} refers to t_1 and t_{latter} to t_2 in the above discussion, and function *Succeeds* (which we will shortly describe in more detail) is used to evaluate causal relationship between two specific task occurrences. Furthermore, recall the fact that each intermediate predicate mandates at least one occurrence of the task to which the Every/Any semantics is applied (e.g. t_1 in the above discussion) within a process execution. Note that it is necessary to explicitly specify a negative result upon the absence of such task occurrences (i.e. $W = \varnothing$) in the evaluation of an *-Every* predicate (and this is not the case for evaluation of an *-Any* predicate)[8].

Let us consider evaluation of the root predicates for causal relationships. Though there are four of them, we consider succession and precedence as two different interpretations of one causal relationship. For example, the statement "task t_1 occurs after task t_2" and the statement "task t_2 occurs before task t_1" refers to one fact that t_1 occurs as result of occurrence t_2. Algorithm 14 specifies the evaluation of intermediate predicate PredEvery, where tasks are treated in a swapped order as compared to the evaluation of SuccEvery.

Finally, we introduce the definitions of two functions *Succeeds* and *ISucceeds*. In Algorithm 15, function *Succeeds* determines if a given e'-event node *eventually* follows a given e-event node in bigraph G (representing a process execution). Following a typical graph search algorithm, it traverses bigraph G from the e-event node (via recursively calling itself) until reaching the e'-event node ($n = m$), the end of the graph (iSuccessors$(G, n) = \varnothing$ where iSuccessors(G, n) denotes the immediate successors of node n in graph G), or a node that was visited before ($n \in V$ where V stores the set of visited nodes). Also, we consider the *successor* relationship is irreflexive, i.e. a task occurrence cannot have a *successor* relationship with itself. Hence, when

[8] By convention, $\forall_{x \in \varnothing} P(x)$ is always true and $\exists_{x \in \varnothing} P(x)$ is always false, regardless of the predicate P(x).

Algorithm 12: Determining the intermediate predicate SuccEvery

function SUCCEVERY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  $W :=$ RetrieveAllEvents(G, t_{former});
  $X :=$ RetrieveAllEvents(G, t_{latter});
  if $W = \varnothing$ **then**
    return FALSE;
  else
    return $\forall_{e \in W} \exists_{e' \in X}$ *Succeeds*$(G, e, e', \varnothing)$

Algorithm 13: Determining the intermediate predicate SuccAny

function SUCCANY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  $W :=$ RetrieveAllEvents(G, t_{former});
  $X :=$ RetrieveAllEvents(G, t_{latter});
  return $\exists_{e \in W} \exists_{e' \in X}$ *Succeeds*$(G, e, e', \varnothing)$

Algorithm 14: Determining the intermediate predicate PredEvery

function PREDEVERY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  $W :=$ RetrieveAllEvents(G, t_{former});
  $X :=$ RetrieveAllEvents(G, t_{latter});
  if $X = \varnothing$ **then**
    return FALSE;
  else
    return $\forall_{e' \in X} \exists_{e \in W}$ *Succeeds*$(G, e', e, \varnothing)$

Algorithm 15: Determining the successor relationship in the bigraph of a conflict-free annotated CFP

function *Succeeds*(G: bigraph, n: node, m: event node, V: set of nodes): boolean
begin
  if $n = m \wedge V = \varnothing$ **then**
    return FALSE;
  else
    if $n = m \wedge V \neq \varnothing$ **then**
      return TRUE;
    else
      if $n \in V \vee$ *iSuccessors*$(G, n) = \varnothing$ **then**
        return FALSE;
      else
        return $\bigvee_{s \in \text{iSuccessors}(G,n)}$ *Succeeds*$(G, s, m, V \cup \{n\})$

e and e' refer to the same task occurrence ($n = m \wedge V = \varnothing$), *Succeeds* returns a negative result. In Algorithm 16, function *ISucceeds* determines if a given e'-event node *immediately* follows a given e-event node in bigraph G. The basic idea is to compute and store the immediate successor event nodes of the e-event node (in S), for each silent event node in S continue the above computation until reaching non-silent event nodes, store the non-silent event nodes (in Y) along the computation, and then check if the e'-event node presents in Y. Also, the *ISucceeds* relationship is irreflexive.

Algorithm 16: Determining the immediate (non-silent) event successor relationship in the bigraph of a conflict-free annotated CFP

function *ISucceeds*(G: bigraph, e: event node, e': event node): boolean
begin
  if $e = e'$ **then**
    return FALSE;
  else
    $Y := \varnothing$;
    K := GetSilentEventNodes(G);
    $S := \bigcup_{c \in iSuccessors(G,e)}$ iSuccessors(G, c);
    while $S \neq \varnothing$ **do**
      select $s \in S$;
      if $s \notin K$ **then**
        $Y \cup := \{s\}$;
      else
        $S \cup := \bigcup_{c \in iSuccessors(G,s)}$ iSuccessors(G, c);
      $S \setminus := \{s\}$;
    return $e' \in Y$

Based on the above discussions, it is straightforward to specify the algorithms for the remaining predicates which are presented in Section Appendix.

3.6 Complexity Analysis

Here we discuss the complexity of generating annotated CFPs and deriving bigraphs from these, and the complexity of evaluating a query.

During preprocessing, we first generate a CFP from a Petri net, and then from the CFP we extract one of more bigraphs. As we only add link information in an annotated CFP, the complexity of the adapted CFP generation algorithm (cf. Algorithm 1) is the same as that of the original CFP algorithm, which is exponential on the number of arcs of the Petri net [6]. The complexity of generating a bigraph from a CFP (cf. Algorithm 2) is linear on the size of the CFP, since the latter is traversed depth-first in reverse order (i.e. starting from a leaf condition).

A basic predicate is evaluated by traversing breadth-first each bigraph of each process model in the repository, thus this operation is linear on the size s of a bigraph. Let b be the total number of bigraphs in the repository and p be the number of basic predicates in a query. Hence, the complexity of evaluating a single query (cf. Algorithms 6–14) is linear on p times b times max_s, where max_s is the size of the largest bigraph in the repository.

It should be noted that for our purposes the adapted CFP generation algorithm and bigraph extraction algorithm are applied to computing the basic predicates over a repository of process models specified as Petri nets. Hence, these operations are performed when inserting a Petri net in the repository. This means that the cost of evaluating a query is not determined by the complexity of these two algorithms, as the computation of the basic behavioural relations would already have taken place (so essentially we trade space for time).

4 Evaluation

In this section we first describe the implementation of APQL in a software tool, and then we report on the performance of APQL which we measured using this tool.

4.1 Implementation

In order to evaluate the performance of APQL we implemented a tool, namely APQL Querier, that supports querying business process model repositories with APQL. The tool is part of the BeehiveZ toolset v3.0. BeehiveZ is an open-source BPM analysis system based on Java (BeehiveZ can be downloaded from `http://code.google.com/p/beehivez/downloads/list`).

The architecture of the APQL Querier and of the Process Model Repository with which the APQL Querier interacts inside BeehiveZ is illustrated in Fig. 9. The core of the APQL Querier is the Query Engine: it takes as input the queries produced by users via the Query Editor and generates as output the querying results via the Query Results Display. The Query Editor uses a concrete syntax of APQL which is defined based on the abstract syntax provided in Section 2.1. This concrete syntax is reported in Appendix.

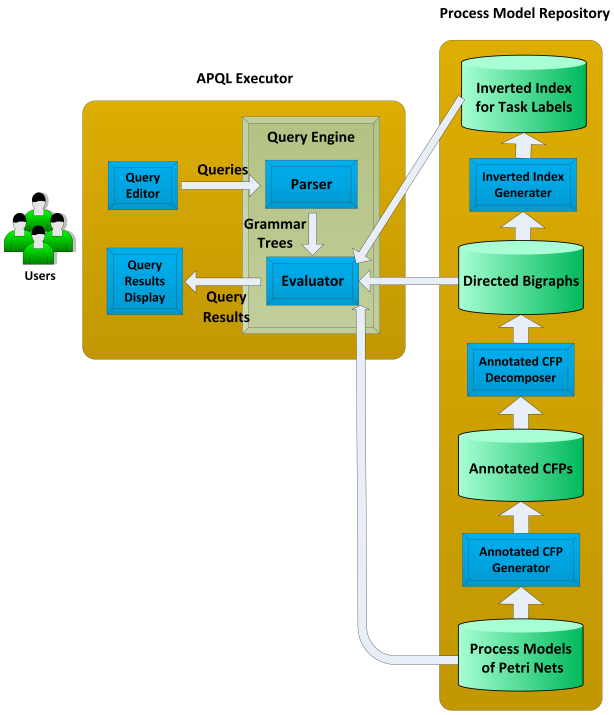

Fig. 9: BeehiveZ: architecture of APQL Querier and Process Models Repository.

Under the hoods, the Query Engine exploits an internal Parser which converts each query statement into a grammar tree. Grammar trees are then used by the Evaluator to identify all process models in the repository that satisfy the requirements of a given query. To do so, the Evaluator needs to get access to the collection of process models stored in the Process Model Repository in Petri net format, as well as to the behavior relation matrices among the events in CFPs. The latter are internally represented as directed bigraphs which have been constructed from the annotated CFPs and relation matrixes of each Petri net by the Annotated CFP Decomposer using Algorithm 2. The generation of annotated CFPs and relation matrixes is in turn performed by the Annoated CFP Generator using Algorithm 1. The data structure of an annotated CFP is based on the underlying incidence matrixes of the original nets. Conditions, events and directed arcs are represented by nodes of doubly-linked lists which support in particular fast insertion of nodes and backward traversing.

Moreover, for efficiency reasons, we keep an inverted index for every node label that appears in the set of annotated CFPs. We use Apache Lucene to manage these indexes.[9] Specifically, for each label we record all processes which contain that label in some node. Based on this index, after a query is issued the tool can instantly filter out a set of candidate models containing the labels used in the query. The rest of the models are

[9] `http://lucene.apache.org`

thus ignored since they are not relevant to the current query. This step typically reduces the scope of searching and increases the tool's performances. Furthermore, an advantage of using inverted indexes is that they can be easily updated as a result of changing a node label in the repository. For more details on this index, we refer to previous work [18].

As an example, we now illustrate how the sample query q_{10} described in Section 2 is executed over the repository of Fig. 4. Let us assume we have created this query through the Query Editor and submitted it to the Query Engine. The Parser is invoked to generate the grammar tree shown in Fig. 3(j). Next, the Evaluator first retrieves the subset of this repository which is relevant to q_{10} using the inverted index on labels. This comprises models 7–12 which include all task labels in q_{10}, i.e. "H", "B" and "C"; the rest of the repository is ignored. Let us take model 11 as an example. Based on the grammar tree, the Evaluator binds all the immediate predecessors of task "H" to variable x (i.e. task "F"), and all the immediate successors of both tasks "B" and "C" to variable y (i.e. task "N"), under the *Assignments* node of the grammar tree. After this, the Evaluator deals with the part of the tree under node *Predicate*. After identifying the task relation $Elt\_TaskSet\_TaskSet$, the Evaluator performs the evaluation of predicate PosSuccAny, according to Algorithm 18, with input of each individual task in x (t_{latter}) and each individual task in y (t_{former}) for each of the process models 7–12 (r). According to the evaluation result, the predicate holds in only model 11 (where task "H" succeeds task "N"). Hence, the Evaluator returns model 11 as the result of q_{10} to the Query Results Display.

4.2 Performance Measurements

We prepared a set of ten sample queries using various APQL predicates, and measured the evaluation of each of these queries over three process model collections. The first two collections are real-life repositories: the SAP R/3 reference model, consisting of 604 EPC models, and the IBM BIT library, consisting of 1,128 Petri nets. The SAP dataset is used by SAP consultants to deploy the SAP enterprise resource planning system within organizations [19]. The IBM BIT library includes five collections (A,B1,B2,C1,C2) of process models from various domains, including insurance and banking [20]. The third dataset contains 100,000 artificially-generated models.[10]

Since the SAP dataset is represented in the EPC notation, we first transformed these models into Petri nets using ProM.[11] This resulted in 591 Petri nets for the SAP dataset (13 SAP reference models could not be mapped into Petri nets through ProM). In the resulting dataset there are 4,439 transitions out of which 1,494 are uniquely-labeled (33% of the total) while in the IBM dataset there are 9,083 transitions with 946 uniquely-labeled one (10% of the total). The structural characteristics of the three datasets used in the experiments are reported in Table 1. In particular, we can see that the SAP and IBM collections have models of comparable sizes based on the average number of their elements (transitions, places, arcs).

Dataset	Models	Transitions	Unique transitions	Avg transitions	Avg places	Avg arcs
SAP	591	4,439	1,494 (33%)	7.5	12.7	19.7
IBM	1,128	9,083	946 (10%)	8.06	10.97	21.47
AG	100,000	248,493	62 (0.026%)	25.06	16.81	63.22

Table 1: Structural characteristics of the three datasets (AG = Artificially Generated dataset).

We generated the third dataset using BeehiveZ based on the reduction rules from [8]. The number of nodes per model follows a normal distribution. Specifically, the number of transitions per model ranges from 1 to 50 (average 25.06), the number of places from 1 to 47 (average 16.81), and the number of arcs from 2 to 162 (average 63.22). The labels of transitions were randomly chosen from a fixed label set comprising the characters "A-Z" and "a-z" and the numbers "0-9", each label being made by a single character or number. In total, this led to 248,493 transitions in this dataset, with 62 unique labels (corresponding to 0.026% of the total number of transitions). We chose such a very low set of unique labels compared to the total number of

[10] This dataset is available at `http://code.google.com/p/beehivez/downloads/list`.

[11] `wwww.processmining.org`

transitions in order to increase the number of models that can potentially satisfy a query, thus simulating the effects of label similarity, which was not implemented. All models used in the experiments were bounded Petri net, which is a requirement for unfolding according to [21].

We conducted our tests on an Intel Core i72600 @3.4GHz and 8GB RAM, running Windows 7 ultimate and JDK6. The heap memory for the JVM was set to 1GB. We executed each query twelve times, and measured each response time. We then discarded the highest and lowest response times for each query and computed the average response time over the remaining ten values. The test queries and the response times for the three datasets are reported in Table 2.

Queries		**Candidate models**			**Returned models**			**Response time [ms]**		
		SAP	**IBM**	**AG**	**SAP**	**IBM**	**AG**	**SAP**	**IBM**	**AG**
q_{11}	List all processes where task [x_1] occurs	5	2	3674	5	2	3674	2.7	51	84
q_{12}	List all processes where task [x_1] occurs before [x_2]	1	1	552	1	1	247	4.5	186	361
q_{13}	List all processes where in every process instance task [x_1] occurs before task [x_2]	3	1	552	1	1	58	12.0	283	609
q_{14}	List all processes where in every process instance every execution of task [x_1] occurs before an execution of task [x_2]	1	1	540	1	0	39	12.2	264	781
q_{15}	List all processes where in every process instance every execution of task [x_1] occurs after an execution of task [x_2]	1	1	593	1	0	33	13.8	249	635
q_{16}	List all processes where task [x_1] occurs in every process instance	5	2	4001	1	1	325	7.2	112	352
q_{17}	List all processes where task [x_1] concurs with task [x_2]	1	1	603	1	0	22	17.9	296	1463
q_{18}	List all processes where either task [x_1] or task [x_2] occurs	1	2	549	1	0	18	17.4	337	1302
q_{19}	List all processes where in every process instance the immediate successors of task [x_1] include task [x_2] and task [x_3]	1	1	76	1	0	4	12.6	415	1490
q_{20}	List all processes where the immediate successors of task [x_1] precedes all of the immediate predecessors of an execution of [x_2]	1	1	587	1	0	11	27.3	563	2306

Table 2: Response times to execute ten sample queries over the three datasets.

The sample queries use various APQL predicates ranging from simple (q_{11}) to complex ones (q_{20}). In particular, q_{11} and q_{12} are used to test the unary predicate *exists*, q_{13} to q_{16} are for *causal relation* predicates, q_{17} and q_{18} are for the *concur* and *exclusive* predicates, while and q_{19} and q_{20} test complex requirements involving variables and set operations. For readability, in the table we use text statements to describe each query and fictitious labels for transitions (e.g. x_1). The corresponding queries written in a concrete syntax of APQL, including the mapping between fictitious labels and real labels from the three datasets, can be found in the Appendix.

The second and third columns of Table 2 show for each query the number of models being filtered by BeehiveZ's inverted index ("candidate models"), and the number of models that actually satisfy the query ("returned models"). These numbers are very low for the SAP and IBM datasets (e.g. q_{13} yields three models in the SAP dataset, out of which only one satisfies the query), due to the high number of unique labels within these collections (see Table 1). However, as expected, these numbers grow significantly in the artificially-generated collection (as an example, q_{13} yields 552 models of which 58 satisfy the query).

The last column of Table 2 shows the response times to execute the sample queries. These times are in the order of milliseconds for the SAP and IBM datasets (average 12.76ms and 276ms) and less than one second for the artificial dataset (average 938ms). This shows that the technique is highly scalable to

very large datasets. Having said that, our technique shifts computation time from query execution to model insertion. In other words, most of the time is employed in generating the CFPs rather than in executing the queries. Specifically, the overall time for building the set of CFPs and the corresponding bigraphs for the three datasets is 11.4 mins (SAP dataset), 26.3 mins (IBM) and 7.6 hours (artificial dataset). However, since we build annotated CFPs incrementally as we insert each Petri net into the repository, in practice the time for creating a single CFP is very short: only 1.15s on average for a model from the SAP dataset, 1.39s for a model from the IBM dataset, and 2.74s for a model from the artificial dataset. These times are reasonable since repository users typically insert or remove single process models, or small groups thereof, at once, rather than inserting or removing entire model collections at once.

As expected, the storage size of the CFPs (including the label indexes) and corresponding bigraphs can be large. While it is only 26.8MB for the SAP dataset and 18.1MB for the IBM dataset, this value gets to 3.38GB for the artificial dataset. However this space is still acceptable considering that in organizational settings dedicate servers are typically employed to host process model repositories, rather than single desktop machines.

5 Related work

Mindful of the importance of query languages for business process models, the Business Process Management Initiative (BPMI) proposed to define a standard process model query language in 2004[12]. While such a standard has never been published, two major research efforts have been dedicated to the development of query languages for process models. One is known as BP-QL [3], a graphical query language based on an abstract representation of BPEL and supported by a formal model of graph grammars for processing of queries. BP-QL can be used to query process specifications written in BPEL rather than possible executions, and ignores the run-time semantics of certain BPEL constructs such as conditional execution and parallel execution.

The other effort, namely BPMN-Q [1, 22], is also a visual query language which extends a subset of the BPMN modelling notation and supports graph-based query processing. Similarly to BP-QL, BPMN-Q only captures the structural (i.e. syntactical) relationships between tasks, and not their behavioral interrelationships. In [23], the authors explore the use of an information retrieval technique to derive similarities of activity names, and develop an ontological expansion of BPMN-Q to tackle the problem of querying business processes that are developed with different terminologies. A framework of tool support for querying process model repositories using BPMN-Q and its extensions is presented in [24].

APQL provides three distinguishing features compared to the above languages. First, its abstract syntax and semantics have been purposefully defined to be independent of a specific process modelling language (such as BPEL or BPMN). This allows APQL and its query evaluation technique to be implemented for a variety of process modelling languages. Second, APQL can express all possible temporal-ordering relations (precedence/succession, concurrence and exclusivity) between individual tasks, between an individual task and a set of tasks as well as between different sets of tasks. Third, these reach querying constructs are evaluated over the execution semantics of process models, rather than their structural relationships. In fact, structural characteristics alone are not able to capture all possible order relations among tasks which can occur during execution, in particular with respect to cycles and task occurrences (recall the discussions in Section 3).

In earlier work [25], we provided an initial attempt at defining a query language based on execution semantics of process models. The language was written in linear temporal logic (LTL) and only supported precedence/succession relations among individual tasks (not sets of tasks).

In addition to the development of a specific process model query language, other techniques are available in the literature which can be useful for querying process model repositories. In [26, 27] the authors focus on querying the content of business process models based on metadata search. VisTrails system [28] allows users to query scientific workflows by example and to refine workflows by analogies. WISE [29] is a workflow information search engine which supports keyword search on workflow hierarchies. In [30] the authors use graph reduction techniques to find a match to the query graph in the process graph for querying process variants, and the approach however works on acyclic graphs only. In [31–33], a group of similarity-based techniques have been proposed which can be used to support process querying. In previous work, we designed

[12] http://www.bpmi.org/downloads/BPMI_Phase_2.pdf

a technique to query process model repositories based on an input Petri net [18]. Finally, in [34], the notion of behavioural profile of a process model is defined, which captures dedicated behavioural relations like exclusiveness or potential occurrence of activities. However, these behavioural relations are derived from the structure of a process model. Thus, for the reasons mentioned above, behavioral profiles only provide an approximation of a process model's behavior, whereas we can precisely determine whether a process model satisfies or not a given query, since we work at the behavioral level. Moreover, the approach requires process models to be sound free-choice Petri nets, whereas our query evaluation technique only requires Petri nets to be bounded, in order to unfold them.

6 Conclusions

This paper contributes an innovative language, namely APQL, for querying process model repositories. APQL provides three main advantages over the state of the art. First, the language is very expressive since it allows users to specify all possible order relationships among tasks or sets thereof. Second, the language is precise, since APQL queries are evaluated over process model behavior, while existing query languages only support structural process characteristics. Third, the language's syntax and semantics are defined independently of any specific process modeling language. APQL is equipped with a technique to evaluate APQL queries. While this query evaluation technique has been designed on top of Petri nets, we demonstrated in the experiments that APQL can be effectively used to query process model repositories defined in other languages, such as EPCs and BPMN. Thus, the technique is generalizable.

The language and its query evaluation technique have been implemented as a component of the process analysis tool BeehiveZ, and evaluated over three large datasets. These include two real-life datasets and a third one which was artificially generated with the specific intent of testing the performance of the implementation against very large datasets. Indeed, the performance measurements show that the technique can efficiently cope with very large datasets (the artificial collection counted 100,000 process models).

Currently APQL only focuses on the control flow perspective of business process models. In the future, we will extend the language definition in order to include other process perspectives such as data and participating resources. Moreover, we plan to run structured interviews with domain experts to assess the overall ease of use and usefulness of APQL.

Acknowledgements

Song and Wang are supported by the National Basic Research Program of China (2009CB320700), the National High-Tech Development Program of China (2008AA042301), the Project of National Natural Science Foundation of China (90718010), and the Program for New Century Excellent Talents in University of China. ter Hofstede and La Rosa are supported by the ARC Linkage Grant "Facilitating Business Process Standardization and Reuse" (LP110100252). In 2010 and 2011, ter Hofstede was a senior visiting scholar of Tsinghua University. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. A. Awad. BPMN-Q: A language to query business processes. In M. Reichert, S. Strecker, and K. Turowski, editors, *Enterprise Modelling and Information Systems Architectures - Concepts and Applications , Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA'07), St. Goar, Germany, October 8-9, 2007*, volume P-119 of *LNI*, pages 115–128. GI, 2007.
2. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 343–354. ACM, 2006.
3. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes with BP-QL. *Inf. Syst.*, 33(6):477–507, September 2008.

4. OMG. *Business Process Model and Notation (BPMN) ver. 2.0*, January 2011. http://www.omg.org/spec/BPMN/2.0.
5. K. L. McMillan. A technique of state space search based on unfolding. *Form. Method. Syst. Des.*, 6(1):45–65, 1995.
6. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Form.Method.Syst.Des.*, 20(3):285–310, 2002.
7. B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990.
8. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, 1989.
9. M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains. In G. Kahn, editor, *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979*, volume 70 of *Lecture Notes in Computer Science*, pages 266–284, London, UK, 1979. Springer-Verlag.
10. J. Engelfriet. Branching processes of petri nets. *Acta Inf.*, 28(6):575–591, 1991.
11. W. M. P. van der Aalst. The application of Petri nets to workflow management. *J. Circuit. Syst. Comp.*, 8(1):21–66, 1998.
12. W. M. P. van der Aalst. Petri-net-based workflow management software. In A. Sheth, editor, *Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems*, pages 114–118. Citeseer, 1996.
13. Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information & Software Technology*, 50(12):1281–1294, 2008.
14. Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.
15. Niels Lohmann. A feature-complete petri net semantics for ws-bpel 2.0. In *Web Services and Formal Methods, 4th International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007. Proceedings*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2007.
16. W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639Ű650, 1999.
17. A. Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, Germany, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1998.
18. T. Jin, J. Wang, N. Wu, M. La Rosa, and A.H.M. ter Hofstede. Efficient and accurate retrieval of business process models through indexing - (short paper). In Robert Meersman, Tharam S. Dillon, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2010 - Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I*, volume 6426 of *Lecture Notes in Computer Science*, pages 402–409. Springer, 2010.
19. G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
20. Dirk Fahland, Cédric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Instantaneous soundness checking of industrial business process models. In *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*, volume 5701 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2009.
21. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106, London, UK, 1996. Springer-Verlag.
22. A. Awad, G. Decker, and M. Weske. Efficient compliance checking using BPMN-Q and temporal logic. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings*, volume 5240 of *Lecture Notes in Computer Science*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.
23. A. Awad, A. Polyvyanyy, and M. Weske. Semantic querying of business process models. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, September 15-19, 2008, Munich, Germany*, pages 85–94. IEEE Computer Society, 2008.
24. S. Sakr and A. Awad. A framework for querying graph-based business process models. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 1297–1300, New York, NY, USA, 2010. ACM.
25. L. Song, J. Wang, L. Wen, W. Wang, S. Tan, and H. Kong. Querying process models based on the temporal relations between tasks. In *Workshops Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference, EDOCW 2011, Helsinki, Finland, August 29 - September 2, 2011*, pages 213–222. IEEE Computer Society, 2011.
26. J. Vanhatalo, J. Koehler, and F. Leymann. Repository for business processes and arbitrary associated metadata. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 426–431, Vienna, Austria, 2006. Springer.

27. A. Wasser, M. Lincoln, and R. Karni. ProcessGene Query - a tool for querying the content layer of business process models. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 1–8, Vienna, Austria, 2006. Springer.
28. C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with vistrails. In J. T. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1251–1254, New York, NY, USA, 2008. ACM.
29. Q. Shao, P. Sun, and Y. Chen. Wise: A workflow information search engine. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1491–1494. IEEE, 2009.
30. R. Lu and S. W. Sadiq. Managing process variants as an information resource. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 426–431, Vienna, Austria, 2006. Springer.
31. Wil M. P. van der Aalst, Ana Karla A. de Medeiros, and A. J. M. M. Weijters. Process equivalence: Comparing two process models based on observed behavior. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2006.
32. M. Ehrig, A. Koschmider, and A. Oberweis. Measuring similarity between semantic business process models. In John F. Roddick and Annika Hinze, editors, *Conceptual Modelling 2007, Proceedings of the Fourth Asia-Pacific Conference on Conceptual Modelling (APCCM2007), Ballarat, Victoria, Australia, January 30 - February 2, 2007, Proceedings*, volume 67 of *CRPIT*, pages 71–80, Darlinghurst, Australia, 2007. Australian Computer Society, Inc.
33. B. Dongen, R. Dijkman, and J. Mendling. Measuring similarity between business process models. In Z. Bellahsene and M. Léonard, editors, *Advanced Information Systems Engineering, 20th International Conference, CAiSE, Montpellier, France, June 16-20, 2008, Proceedings*, volume 5074 of *Lecture Notes in Computer Science*, pages 450–464, Berlin, Heidelberg, 2008. Springer-Verlag.
34. M. Weidlich, J. Mendling, and M. Weske. Efficient consistency measurement based on behavioral profiles of process models. *Software Engineering, IEEE Transactions on*, 37(3):410 –429, 2011.

Appendix

Here we provide an example concrete syntax for the APQL language. Next, we show an instantiation of this syntax for the ten sample queries used in the experiments in Table 3, while in Table 4 we provide the mapping between the fictitious labels used in Table 2 and the real labels used in the SAP, IBM and artificial datasets. Finally, the algorithms of the basic predicates, AlwSuccAny, PosSuccAny, AlwISuccEvery, AlwISuccAny, PosISuccEvery, PosISuccAny, AlwPredEvery, AlwPredAny, PosPredEvery, PosPredAny, AlwIPredEvery, AlwIPredAny, PosIPredEvery, PosIPredAny, are presented.

$$
\begin{aligned}
< Query > &::= \text{“\{”} < Assignments > \text{“\}”} < Predicate > \\
< Assignments > &::= < Assignment > [\text{“;”} < Assignment >]^* \mid \text{“ ”} \\
< Assignment > &::= < TaskSetVar > \text{“ = ”} < TaskSet > \\
< TaskSet > &::= \text{“[”} < SetofTask > \text{“]”} \mid < TaskSetVar > \mid < Application > \\
&\quad \mid < Construction > \\
< SetofTask > &::= < Task > [\text{“,”} < Task >]^* \\
< Task > &::= < TaskLabelExpr > \\
< TaskLabelExpr > &::= < TaskLabel > [\text{“ : ”} < SimDegree >?] \\
< Application > &::= < TaskCompOp >< TaskSet >< AnyAll > \\
< TaskCompOp > &::= \text{“}alwpredevery\text{”} \mid \text{“}alwpredany\text{”} \mid \text{“}pospredevery\text{”} \mid \text{“}pospredany\text{”} \mid \\
&\quad \text{“}alwsuccevery\text{”} \mid \text{“}alwsuccany\text{”} \mid \text{“}possuccevery\text{”} \mid \text{“}possuccany\text{”} \mid \\
&\quad \text{“}alwipredevery\text{”} \mid \text{“}alwipredany\text{”} \mid \text{“}posipredevery\text{”} \mid \text{“}posipredany\text{”} \mid \\
&\quad \text{“}alwisuccevery\text{”} \mid \text{“}alwisuccany\text{”} \mid \text{“}posisuccevery\text{”} \mid \text{“}posisuccany\text{”} \mid \\
&\quad \text{“}concur\text{”} \mid \text{“}exclusive\text{”} \\
< Construction > &::= < TaskSet >< Set\_Op >< TaskSet > \\
< AnyAll > &::= \text{“}any\text{”} \mid \text{“}all\text{”} \\
< Set\_Op > &::= \text{“}union\text{”} \mid \text{“}intersection\text{”} \mid \text{“}minus\text{”} \\
< Predicate > &::= < Task > \text{“}exist\text{”} \mid < Task > \text{“}alwexist\text{”} \mid \\
&\quad < TaskRel > \mid < Bin\_Predicate > \mid < Un\_Predicate > \\
< TaskRel > &::= < TaskRelExpr > \\
< TaskRelExpr > &::= < TaskInTaskSet > \mid < Task\_TaskSet > \mid < Task\_Task > \mid \\
&\quad < Elt\_TaskSet\_TaskSet > \mid < Set\_TaskSet\_TaskSet > \\
< TaskInTaskSet > &::= < Task > \text{“}belongto\text{”} < TaskSet > \\
< Task\_TaskSet > &::= < Task >< TaskCompOp >< TaskSet >< AnyAll > \\
< Task\_Task > &::= < Task >< TaskCompOp >< Task > \\
< Elt\_TaskSet\_TaskSet > &::= < TaskSet >< TaskCompOp >< TaskSet >< AnyAll > \\
< Set\_TaskSet\_TaskSet > &::= < TaskSet >< SetCompOp >< TaskSet > \\
< SetCompOp > &::= \text{“}identical\text{”} \mid \text{“}subsetof\text{”} \mid \text{“}overlap\text{”} \\
< Bin\_Predicate > &::= < Predicate >< BinLogOp >< Predicate > \\
< Un\_Predicate > &::= \text{“}not\text{”} < Predicate > \\
< BinLogOp > &::= \text{“}and\text{”} \mid \text{“}or\text{”}
\end{aligned}
$$

	Concrete Queries
q_{11}	x_1 exist
q_{12}	x_1 pospredany x_2
q_{13}	x_1 alwpredany x_2
q_{14}	x_2 alwsuccevery x_1
q_{15}	x_2 alwpredevery x_1
q_{16}	x_1 alwexist
q_{17}	x_1 concur x_2
q_{18}	x_1 exclusive x_2
q_{19}	{x = alwisuccany x_1} x_2, x_3 subsetof x
q_{20}	{x = posisuccany x_1, y=posipredany x_2} x pospredany y

Table 3: The ten example queries used in Table 2 in the concrete syntax of APQL.

Queries	Labels	SAP	IBM	AG
q_{11}	x_1	Customer Quotation Processing	process.s0000009##s000001827.outputCriterion.s00000859	Z
q_{12}	x_1,	Subsequent Acquistion,	process.s00000247##s00002258.outputCriterion.s00000772,	M,
	x_2	Processing of Asset Acquistion	process.s00000266##s00002468.outputCriterion.s00000773	N
q_{13}	x_1,	Customer Quotation Processing,	process.s00000285##s00002171.outputCriterion.s00000743,	M,
	x_2	Sales Order Processing	process.r00000266##n00002468.outputCriterion.s00000773	N
q_{14}	x_1,	Order Release,	process.s00000202##s00001694.outputCriterion.s00000773,	L,
	x_2	Cost Reposting	process.r00000251##n00004029.outputCriterion.s00000471	S
q_{15}	x_1,	Settlement Account Assignment,	process.s00000243##s00002261.outputCriterion.s00005267,	J,
	x_2	Periodic Settlement	process.r00000106##n00002617.outputCriterion.s00000704	W
q_{16}	x_1	Customer Quotation Processing	process.s00000275##s00002184.outputCriterion.s00000838	K
q_{17}	x_1,	Subsequent Acquistion,	process.s00000247##s00002258.outputCriterion.s00000772,	A,
	x_2	Processing of Asset Acquistion	process.s00000266##s00002468.outputCriterion.s00000773	H
q_{18}	x_1,	Product Structure Management for Variant Products,	process.s00000265##s00002061.outputCriterion.s00000774,	M,
	x_2	Product Structure Management via CAD	process.r00000266##s00009387.outputCriterion.s00000721	R
q_{19}	x_1,	Set Deletion Flag,	process.s00000204##s00003268.outputCriterion.s00000859,	C,
	x_2,	Delete without Archiving,	process.r00000473##n00007961.outputCriterion.s00000328,	O,
	x_3	Delete with Archiving	process.r00000663##n00007132.outputCriterion.s00000331	y
q_{20}	x_1,	Specification,	process.s00000754##s00003684.outputCriterion.s00000930,	L,
	x_2	Schedule Update and Confirmation	process.s00000855##n00006324.outputCriterion.s00001064	W

Table 4: Mapping of the task labels used in Table 2 with those in the SAP, IBM and AG datasets.

Algorithm 17: Determining the predicate AlwSuccAny

function ALWSUCCANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *SuccAny*($t_{former}, t_{latter}, G$)

Algorithm 18: Determining the predicate PosSuccAny

function POSSUCCANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *SuccAny*($t_{former}, t_{latter}, G$)

Algorithm 19: Determining the intermediate predicate ISuccEvery

function ISUCCEVERY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  W := RetrieveAllEvents(G, t_{former});
  X := RetrieveAllEvents(G, t_{latter});
  if $W = \varnothing$ **then**
    return FALSE;
  else
    return $\forall_{e \in W} \exists_{e' \in X}$ *ISucceeds*(e, e', G)

Algorithm 20: Determining the intermediate predicate ISuccAny

function ISUCCANY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  W := RetrieveAllEvents(G, t_{former});
  X := RetrieveAllEvents(G, t_{latter});
  return $\exists_{e \in W} \exists_{e' \in X}$ *ISucceeds*(e, e', G)

Algorithm 21: Determining the predicate AlwISuccEvery

function ALWISUCCEVERY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *ISuccEvery*($t_{former}, t_{latter}, G$)

Algorithm 22: Determining the predicate AlwISuccAny

function ALWISUCCANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *ISuccAny*($t_{former}, t_{latter}, G$)

Algorithm 23: Determining the predicate PosISuccEvery

function POSISUCCEVERY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *ISuccEvery*($t_{former}, t_{latter}, G$)

Algorithm 24: Determining the predicate PosISuccAny

function POSISUCCANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *ISuccAny*($t_{former}, t_{latter}, G$)

Algorithm 25: Determining the predecessor relationship in the bigraph of a conflict-free annotated CFP

function *Precedes*(G: bigraph, n: node, m: node, V: set of nodes): boolean
begin
  if $n = m \wedge V = \varnothing$ **then**
    return FALSE;
  else
    if $n = m \wedge V \neq \varnothing$ **then**
      return TRUE;
    else
      if $n \in V \vee$ *iPredecessors*$(G, n) = \varnothing$ **then**
        return FALSE;
      else
        return $\bigvee_{s \in \text{iPredecessors}(G,n)}$ *Precedes*$(G, s, m, V \cup \{n\})$

Algorithm 26: Determining the intermediate predicate PredAny

function PREDANY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  W := RetrieveAllEvents(G, t_{former});
  X := RetrieveAllEvents(G, t_{latter});
  return $\exists_{e' \in X} \exists_{e \in W}$ *Succeeds*$(G, e, e', \varnothing)$

Algorithm 27: Determining the predicate AlwPredEvery

function ALWPREDEVERY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *PredEvery*$(t_{former}, t_{latter}, G)$

Algorithm 28: Determining the predicate AlwPredAny

function ALWPREDANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *PredAny*$(t_{former}, t_{latter}, G)$

Algorithm 29: Determining the predicate PosPredEvery

function POSPREDEVERY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *PredEvery*$(t_{former}, t_{latter}, G)$

Algorithm 30: Determining the predicate PosPredAny

function POSPREDANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *PredAny*($t_{former}, t_{latter}, G$)

Algorithm 31: Determining the immediate (non-silent) event predecessor relationship in the bigraph of a conflict-free annotated CFP

function *IPrecedes*(G: bigraph, e: event node, e': event node): boolean
begin
  if $e = e'$ **then**
    return FALSE;
  else
    $Y := \varnothing$;
    K := GetSilentEventNodes(G);
    $S := \bigcup_{c \in iPredecessors(G,e)}$ iPredecessors(G, c);
    while $S \neq \varnothing$ **do**
      select $s \in S$;
      if $s \notin K$ **then**
        $Y \cup := \{e'\}$;
      else
        $S \cup := \bigcup_{c \in iPredecessors(G,s)}$ iPredecessors(G, c);
      $S \setminus := \{s\}$;
    return $e' \in Y$

Algorithm 32: Determining the intermediate predicate IPredEvery

function IPREDEVERY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  W := RetrieveAllEvents(G, t_{former});
  X := RetrieveAllEvents(G, t_{latter});
  if $X = \varnothing$ **then**
    return FALSE;
  else
    return $\forall_{e' \in X} \exists_{e \in W}$ *ISucceeds*(G, e, e')

Algorithm 33: Determining the intermediate predicate IPredAny

function IPREDANY(t_{former}: taskID, t_{latter}: taskID, G: bigraph): boolean
begin
  W := RetrieveAllEvents(G, t_{former});
  X := RetrieveAllEvents(G, t_{latter});
  return $\exists_{e' \in X} \exists_{e \in W}$ *ISucceeds*(G, e, e')

Algorithm 34: Determining the predicate AlwIPredEvery

function ALWIPREDEVERY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *IPredEvery*($t_{former}, t_{latter}, G$)

Algorithm 35: Determining the predicate AlwIPredAny

function ALWIPREDANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigwedge_{G \in \mathbb{G}}$ *IPredAny*($t_{former}, t_{latter}, G$)

Algorithm 36: Determining the predicate PosIPredEvery

function POSIPREDEVERY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *IPredEvery*($t_{former}, t_{latter}, G$)

Algorithm 37: Determining the predicate PosIPredAny

function POSIPREDANY(t_{former}: taskID, t_{latter}: taskID, r: process model): boolean
begin
  $\mathbb{G}$:= RetrieveBigraphs(r);
  return $\bigvee_{G \in \mathbb{G}}$ *IPredAny*($t_{former}, t_{latter}, G$)